PROCEEDINGS

of the

Fourth Computer Graphics Workshop

Cambridge, MA, 1986 7

USENIX Association

Fourth Computer Graphics Workshop

Cambridge, MA, 1987

PROCEEDINGS

October 8-9, 1987

For additional copies of these proceedings, or copies of the
Proceedings of the First, Second or Third Computer Graphics Workshops, write

USENIX Association
P.O. Box 2299
Berkeley, CA 94710 USA

The price per copy of the Third and Fourth Proceedings is $10.00,
plus $15 for overseas (air) postage.

The price per copy of the First and Second Proceedings is $3.00,
plus $7 for overseas (air) postage.

ACKNOWLEDGMENTS

Sponsored by:	USENIX Association P.O. Box 2299 Berkeley, CA 94710	
Program Chairs:	Tom Duff Lou Katz	AT&T Bell Laboratories Metron Computerware, Ltd.
Program Committee:	Reidar J. Bornholdt Tom Duff Michael Hawley Lou Katz	Columbia University AT&T Bell Laboratories MIT Media Lab Metron Computerware, Ltd.
Workshop held at:	Boston Marriott Cambridge Cambridge, Massachusetts	
USENIX Meeting Planner:	Judith F. DesHarnais	
Proceedings Production:	Peter H. Salus Tom Strong	USENIX Executive Director Strong Consulting

TABLE OF CONTENTS

More Music Software for Unix

Michael Hawley
MIT Media Lab
Cambridge, MA

We are trying to write programs which can analyze music performance data (e.g., MIDI) and make sense of it. For instance, a machine should be able to "listen" to someone playing a 4-voice Bach chorale and do what a harmony student or musical idiot-savant might do - write down the notes correctly, determine the key signature and meter, determine where to draw bar lines and separate the voices into streams. The current programs try to do a kind of multidimensional clustering of the data: music is "chunked" up in a variety of ways (for instance, by making aggregates of notes whose attacks are close in time, like chords or fast scales; or by doing a 2d clustering in pitch-time space - that is, connecting the nearest dots on a piano roll). As many chunks as possible are hashed into a database in several dimensions - we might hash groups by melodic slope, harmonic outline, rhythmic quantization, duration, etc - and when collisions occur, an association is made between the colliding groups, and a differencing algorithm can then relate similar groups by noting some measure of their distance (say, "these figures are exactly the same but one is transposed down a fifth).

After enough of a database has been built up one can consider asking other kinds of questions: "do you recognize this piece? does it sound like Bach or like Mozart? have you heard anything like this melody before?"

The hardware is currently a Sun-3 which controls 4 mpu-401's (theoretically permitting 64 channels of MIDI i/o). The Bosendorfer interface may be completed over the summer.

Putting It All Together: An Integrated Package for Viewing and Editing 3D Microworlds

Cliff Brett, Steve Pieper and David Zeltzer
Computer Graphics and Animation Group
The Media Laboratory
Massachusetts Institute of Technology
Cambridge, MA 02139

ABSTRACT

BOLIO is a system under development with two major design goals: to provide an intuitive graphical editor at multiple levels of detail, and to serve as a modular and device-independent interface for a variety of applications. We describe the representation and organization of graphical objects — including menus and viewports — and the interactive control strategy that we have implemented to provide the requisite extensibility and generality. As an editor, BOLIO allows interactive modification of polygonal objects by operating on points, polygons and edges. In addition, whole objects can be transformed, positioned and oriented; assembled into articulated, composite objects; or specified as components of a larger 3D scene. Finally, BOLIO incorporates a constraint toolkit to allow descriptions of the natural relationships among objects in a scene, e.g., "Put the cup on the table," and other geometric and physical invariants (e.g., gravity). As a general viewing and transformation package, BOLIO can serve as a front end for a variety of applications, including an object modeler based on generalized cylinders, and a 3D figure animation system.

1. Introduction

A long-term objective of our research is the design and implementation of a *task level* animation system which allows the specification of behaviors implicitly, in terms of events, constraints and rather abstract motor goals. (See Zeltzer[1] for a taxonomy and survey of computer animation systems). We view computer animation as a window on a virtual microworld, with which the user can interact — ultimately in realtime. We therefore need interactive, graphical tools to view and manipulate objects at all stages of the graphics pipeline: object creation and editing, assembling scenes and composite objects, describing and simulating behaviors, viewing and rendering. Our aim in the development of BOLIO has been to design and implement a robust and consistent graphical front-end for an animation environment. Such a front-end would serve a graphical function analogous to *emacs*, for example, which provides a habitable environment for program development, testing and debugging; or to a *browser* in an object lattice, which allows examination and modification of classes, subclasses and instances.

The notion of a modular software system is well-established in computer graphics[2-4], and is usually referred to as a *software testbed* for developing graphical systems. Earlier work on graphics testbeds, as well as the recently reported GRAPE[5] and FRAMES[6] systems have been primarily

Authors' address: MIT Media Lab, 20 Ames Street, Cambridge, MA 02139.

e-mail: {cliff|pieper|dz}@media-lab.mit.edu

This work was supported in part by NHK (Japan Broadcasting Corp.), CPW Technology, and an equipment grant from Hewlett-Packard, Inc.

concerned with the development and testing of rendering programs, or with easily-composable UNIX† pipes for viewing single frames, rather than interaction with simulations. Parent[7], and later Ressler[8], describe early experiments with interactive object editors for realtime display; the functionality and interface styles of both systems are similar to the techniques used for editing polyhedra in BOLIO. MacDougal, Gomez and Zeltzer[9], describe a standalone interactive polyhedral object editor as a component of an animation environment — which included Crow's *scn_assmblr* rendering system[3] — at Ohio State University.

While BOLIO started out as an object editor, we quickly realized that many other applications share the same interface requirements. In addition to editing attributes of polyhedral objects, the BOLIO system has therefore evolved to allow experimental application of geometric and behavioral modeling tools in an interactive setting. Users can pursue the two tasks independently, through a uniform user interface. BOLIO includes a solid modeler, *Pathtool,* based on generalized cylinders, so that data can be created within BOLIO. Objects that have been generated with other modelers can be input from files in a standard data format. However derived, objects can then be edited in terms of their primitives to create a final desired shape.

Behavior modeling tools include a constraint mechanism which controls the reaction of agents to simulated changes in the virtual environment, and inverse kinematics routines. Since these tools have been integrated into the BOLIO environment, they can manipulate the shared database of microworld information without specific interface code.

As a generalized graphical front end, BOLIO requires a standard program interface, a uniform and consistent user interface, and a standard data representation. Our polyhedron data representation is a variant of that described in Crow[3], and is standardized across machines and programs — i.e., objects generated via solid modeling tools on a lisp machine can be accessed via ethernet and input by BOLIO on a Hewlett-Packard 9000 series workstation. The next section looks at the internal design of BOLIO, including command interface, i/o facilities and internal data structures. Section 3 outlines the facilities for assembling and editing polyhedral objects. In Section 4 we discuss the constraint package.

2. Modularity, Loose-coupling and Device-Independence

2.1. BOLIO's Main Loop

BOLIO maintains and continually traverses three fundamental lists: active viewports (vp_world), graphical objects (bobj_world), and the command stack. By modifying, adding and deleting items from these lists, BOLIO — and integrated application programs — control the display and updating of all graphical objects. Viewport and object lists are discussed in the section on Data Structures, below. This section describes the use of the command stack.

BOLIO's command loop is table driven, with an internally managed stack. An entry on BOLIO's command stack is an index into an array of command structures. These command structures have function pointers to the code which executes the command, as well the command name in string form. New application modules are incorporated into the system simply by adding their command structure to the command array. The function pointer in the command structure points to an *execute* function for the command.

The main loop first reads the input state (from the binput data structure, discussed below), then performs hit-testing and picking functions for the cursor sensitive areas of the screen. It then invokes the *execute* function for the top command on the command stack, and repaints any parts of the screen which have become "dirty" (marked for update) as a result of executing the command. The *execute* function returns one of three values: pending, finished, or error. If the return code is finished, the command index is popped from the command stack. If the return code is error, the command index is popped and an error message is printed. If the return code is pending, the command is left on the stack, and the main loop goes through another iteration. This method was

† UNIX is a trademark of AT&T Bell Laboratories.

chosen because it provides the benefits of standard input processing and screen management without the computational overhead of multiple processes and interprocess communication.

2.2. Representation of Graphical Objects and Viewports

BOLIO can maintain displays with multiple viewports, including adjustable levels of rendering detail, a menu and/or keyboard/script based command interface, and object database management facilities. These form a device-independent core of support so that writers of new applications can concentrate on the unique aspects of the application, without re-implementing display routines or user interaction code.

2.2.1. Bobjects

BOLIO is object-oriented. The generic data structure associated with each graphical object is called a **bobject** (for "BOLIO object"). There are four types of **bobjects**: lights; cameras; polyhedra; and **client** objects, which can be any other kind of object with a non-standard geometric or procedural description. A **bobject** is declared as in Figure 1.

```
typedef struct _bobj {
  char *name;
  LIST *worlds;
  Generic *description;
  LIST *optical_props;
  BEARINGS bearings;
  struct limbs *constraints;
  LIST *drawing_objs;
} bOBJECT;
```

Figure 1. The **bobject** data structure.

[Name.] This is a string which distinguishes this object from all others in the world.

[Worlds.] This is a list of pointers to all the **bobj_worlds** where the object is posted. A **bobj_world** is a list of objects which may appear together in a scene. Any number of objects may exist in a **bobj_world**; objects may exist simultaneously in more than one **bobj_world**. The user can create new **bobj_worlds** or move between existing ones easily. The **bobj_worlds** are collectively known as the **bobj_universe**.

[Description.] This is a generic pointer to a representation of the object, which can be one of four types: light, camera, polyhedron, or some non-standard **client** representation. For geometric objects, the description might be explicit (e.g., points and polygons) or parametric (e.g. splines). Non-geometric objects can be accommodated as well: the description of a camera, for example, specifies viewing parameters. The first byte of the description gives the object's **type**, which tells BOLIO how to interpret the description.

[Optical props.] This is a list of pointers to structures containing the object's optical properties: color, shininess, etc. A single optical description may apply to the entire object, or merely to some small part of it. For example, a user might wish to specify a different color for each polygon or vertex of a polyhedron.

[Bearings.] This is a structure containing information about the object's position and orientation. Positional information includes the object's center, bounding box, and radius (for bounding sphere). Orientation information is specified in terms of two vectors, **up** and **normal**. The bearings structure also contains space for a transformation matrix and a local origin to be used as a transformation center.

Constraints. This is a pointer to the constraints which affect the object. These structures are described in Section 4.

Drawing objs. This is a list of pointers to device-dependent structures which contain all the information necessary to display the object. HP's Starbase supports a variety of graphics primitives: points, lines, polygons, text, bit-block transfers, spline curves and surfaces, arcs and ellipses. This data is maintained in local object coordinates. At display time, the data is transformed using the current transform matrix stored in the **bearings** structure. **The drawing_obj is the only device-dependent portion of the bobject data structure.** There are two options for applications programmers who wish to use **client** (non-standard) object representations. First, non-standard geometric data can be converted to the BOLIO polyhedral format (e.g., by subdividing patches into planar polygons), and BOLIO will handle these objects in the usual way. Alternatively, the applications code can compute the **drawing_obj** directly, using the graphic primitives appropriate for the host machine, and attach it to the **bobject**. At display time, BOLIO can handle such objects as usual. Thus applications that make use of non-standard geometric data can use the **bobject** data structure unchanged, and modifications for porting BOLIO to other hosts are localized.

2.2.2. Viewports.

Since X windows were not usable with the graphics accelerators on our HP workstations, we designed and implemented our own 3D window system. BOLIO viewports emulate X windows in many respects. All standard windowing functions are supported: viewports may be interactively created, destroyed, moved, stretched, hidden, exposed, iconified or de-iconified. Viewports can overlap; objects in farther viewports may pass behind nearer viewports. Users may define a customized viewport environment by creating a configuration file which will be used by the program at startup.

BOLIO manages its viewports in the same way it keeps track of **bobjects**: a **vp_world** contains a group of viewports which are to be displayed together. As with a **bobject**, a viewport may simultaneously exist in more than one **vp_world**. The user can create new **vp_worlds** or move between existing ones easily. All the **vp_worlds** are collected together in a list known as the **vp_universe**

Viewports are only loosely coupled to cameras. Recall that a camera is represented as a **bobject**, just like any other graphical object. A **bobj_world** contains a list of **bobjects**, some or none of which may be cameras. When a viewport is associated with a given **bobj_world**, the list of **bobjects** is searched to see if cameras are present. If there are none, a default camera **bobject** is created for that **bobj_world**. It there is already a single camera, that becomes the current view. If there is more than one camera, the last camera used for this viewport — in this **bobj_world** — is selected. Otherwise one of the cameras is chosen by some rule (first found on the list, most recently used, etc.); it is easy to interactively select any of the other cameras. This allows viewports to be switched among **bobj_worlds** freely, always presenting a consistent view of any world. Once a camera is selected, the viewing parameters are "compiled" and stored — i.e., the various perspective and scaling matrices are computed.

Associated with a viewport is a rendering style — wireframe, smooth-shaded, etc. The rendering mode is usually determined from the **bobject**, but the user can force the viewport mode to override, if for example, one wanted to mix wireframe and smooth-shaded views of the same world. If the rendering style is not listed in the **bobject**, then the viewport style is used. The default mode is set according to the capability of the current host. See Figure 2, which shows in schematic form the organization of objects and viewports. Figure 3 shows a view of a typical BOLIO display, in which several objects have been linked into a kinematic chain and positioned using inverse kinematics (See Section 4).

2.3. I/O Management

The input and output stages of BOLIO incorporate a layer of device-independent data structures which insulate applications code from specific I/O devices. BOLIO provides a single, high-level input routine, **binput_capture_input_state**. This function automatically handles input from *any* currently opened input device — keyboard, mouse, tablet, etc — so that input can be freely

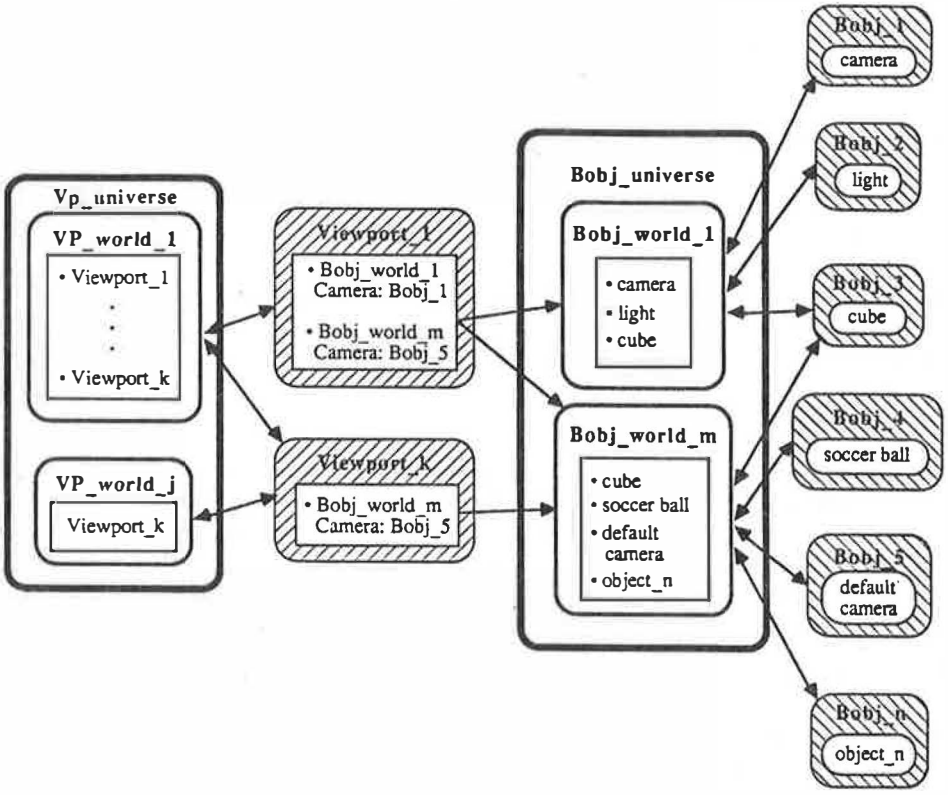


Figure 2. A schematic view of the organization of viewport and object groupings used in BOLIO.

intermixed. **Binput_capture_input_state** sets appropriate values in the **binput** data structure; BOLIO and applications routines acquire input by accessing this structure. Similarly, output is performed by building a data structure which is interpreted by the output routines. This device abstraction technique allows, for example, easy cursor management for device-independent locator input on various machines.

The **binput_capture_input_state** function maintains the string input facilities. The **binput** structure contains a stack of file pointers, with the top pointer indicating the current input file. Input files can be nested arbitrarily with an *#include* facility, which operates much like the C preprocessor. If the top file is the standard input, the keyboard is sampled with a non-blocking read operation. Text from the keyboard or other files is collected into lines which are added to an internal string buffer. Text can also be added to the string buffer through user interaction with the menu system (described below). Any command keywords in the string are detected, and cause the command to be placed on the **command stack** and executed on the next pass through the main loop. When string input is required, applications perform a blocking read on the internal string buffer, which then returns to the main interaction loop until the string buffer is non-empty.

The **binput** structure contains the current state of all the input devices in the system which have been opened for input, as well as the state of the cursor, which is handled as a virtual input device. **Binput_capture_input_state** also performs the update function on the cursor device, in either relative coordinates (e.g., from the mouse), or in absolute coordinates (e.g., from a tablet).

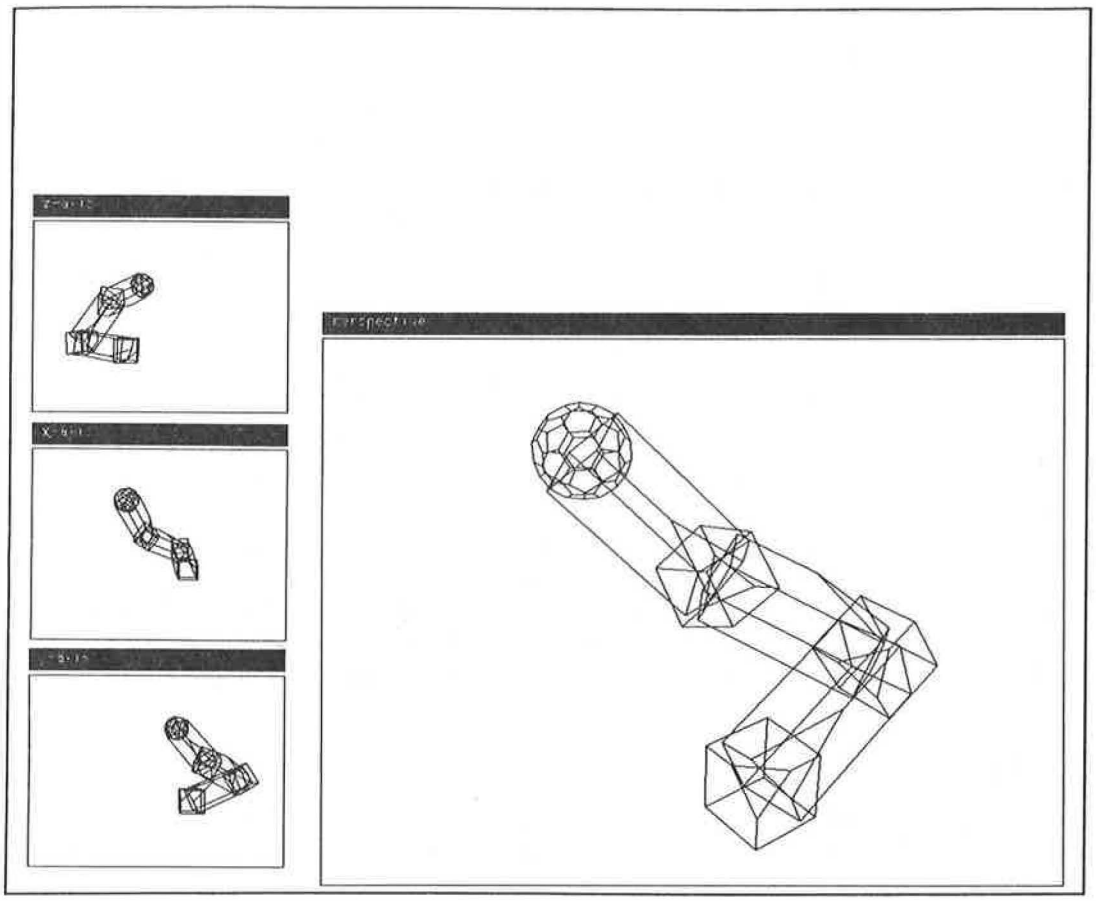


Figure 3. A print of a BOLIO display, showing objects that have been linked into a kinematic chain and have been positioned using inverse kinematics.

Cursor motion can also be controlled from an *#include* script file.

Graphics output is handled automatically by BOLIO through the **bobject's drobject** structures, described in Section 2.2.1. Text output for the non-graphics screen is sent to standard output or standard error with no processing by BOLIO.

2.4. Menus

BOLIO's menu system is device-independent, hierarchical, and string-based. It supports pop-up menus either at an absolute screen location, or relative to the current cursor location. The menu system manages a stack of currently displayed menus, and includes commands for pushing and popping menus, as well as redrawing the entire menu stack. Arbitrary menu lattices are described in ASCII configuration files which allows the association of return strings with individual menu items, and makes it easy to add new commands for menu display. A menu item may additionally have an associated sub-menu, which is automatically displayed when the menu item is selected. BOLIO's main loop performs hit-testing and picking on the menus based on the cursor value supplied by the **binput** module. If the picked menu item has an associated return string, it is passed to the **binput** module which adds it to the string buffer, or parses the string if a command is detected. This menu package has been ported to other applications running on HP and Sun workstations.

2.5. Device Independence

BOLIO is currently implemented on Hewlett-Packard 9000 Series 300 workstations with either the HP 98710 (which supports 3D wireframe graphics in hardware) or HP 98721 SRX (with hardware support for smooth shading) graphics systems. Although these systems both run HP's Starbase graphics package (based on the CGI standard), capabilities of the devices are significantly different in some important respects. Specifically, the 98710 has an 1024x768 frame buffer, with 8-bit pixels and a 24-bit output color look-up table; the 98721 SRX is a 28-bit 1280x1024 frame buffer with 4 of the bits acting as an independent overlay device.

In the 98710 version, the menu system is drawn on the same screen as other graphical objects, while in the 98721 version menus are drawn in the overlay planes, eliminating the need to redraw active menus when the viewports are being updated dynamically. We call the collection of viewports, menus, menu bars, sliders and similar interface objects biobs (for BOLIO input/output objects). BOLIO handles functional device dependence by maintaining sorted lists of active biobs for each open output device. At initialization, each biob is placed on the list corresponding to the device on which it is to appear. When the screen is refreshed, only those biobs on the same device as a "dirty" (marked for display update) element are redrawn. (Since viewports can overlap, we currently redraw all viewports in front of a "dirty" viewport). With this technique, menus can be drawn either in the same device as the viewports, or in the overlay planes, and menu refreshing is managed automatically.

The 98721 SRX also has hardware-assisted shading capabilities. All graphical output is processed through the **drobject** and **viewport** data structures, which have flags selecting the type of rendering preferred for the object and the viewport. (Rendering type defaults to the viewport if not indicated in the **bobject**). If the current device supports the chosen rendering method, the object is drawn that way, otherwise the object is drawn in the best approximation available on the current device.

3. Data Editing.

Currently, only polyhedral object editing is supported, but we expect the range of representations to expand as BOLIO continues to evolve. Geometric operations can be performed graphically and interactively on polyhedral objects at several levels of detail: the user can compose, delete and modify various multi-object groupings; the user can transform and modify individual objects; and it is possible to operate on the constituent geometric primitives of an object. Any **bobject** can be edited, including lights and cameras, although clearly not all operations are defined on all objects. Since objects are typed, it is easy to determine the validity of an operator for a particular object.

First, objects can be instanced and assembled into scenes, transformation groupings (in which all objects are subject to the same transformation matrix), and linkages. As described in Section 4, various constraints, as well as inverse kinematics routines, can be applied to these groupings, which can be named (i.e., as bobj_worlds) and assigned their own viewport or viewports.

Individual objects can be transformed in the usual ways — rotation, translation and so on. In addition, these transformations can be constrained, so that objects can be aligned and attached in various ways, as described in Section 4.

The geometric primitives of an object can also be edited. Operations supported include moving, adding and deleting points, polygons and edges; joining and splitting whole objects as well as individual polygons; and consistency and planarity checking for polygons.

Interactive, gesture-based object editing requires a mechanism for fast 3D hit-testing. In order to do this in a complicated scene, some method is needed to spatially organize the geometric database to minimize searching. We have chosen the octree method because it is adaptive to the complexity of objects in the scene[10].

BOLIO builds a world space octree for each bobj_world. This is a very low-resolution tree — only the position of whole objects (based on their bounding boxes) is stored. If a user wishes to perform hit-testing on smaller elements (e.g., polygon vertices), BOLIO constructs a higher resolution octree for a localized set of objects. (Transformations on objects must be controlled when the octree

is active, of course, to minimize the need for recalculating the octree or portions of it). Our implementation allows arbitrary data types to be stored at any node of the tree, so applications which require new data representations can make use of the octree package.

Our polygonal data representation maintains connectivity information. Polygons, for instance, have pointers to their constituent vertices and edges. This information is kept so that the cursor may be rapidly dragged over the surface of a large, complex object. It is also convenient when changes are made to the structure of an object: BOLIO can quickly determine how the region surrounding the target area of the edit is affected by any changes.

4. Assemblies, Kinematic Chains and Constraints

BOLIO's constraint system was developed to allow the description and maintenance of dependency relationships among objects in a microworld. Dependency relationships are specifications of how the properties of an object are modified as a function of the properties of another object or set of objects. The constraint system provides general tools for assembling networks of constraints, and for systematically satisfying them. Three major applications of constraints are currently implemented in BOLIO: for object positioning, for dynamic simulation, and as a software interface for adding new data types (such as spline surfaces or superquadrics) to the editing environment. We first look at the functionality of the constraint system in terms of positioning and dynamic simulation, then examine how these facilities make the program easy to extend.

A traditional form of positioning relationship employed in computer graphics is the transformation hierarchy, in which composite objects are formed by describing a tree of matrices[11]. The entire composite object can be manipulated by changing the matrix at the top level of the tree, or the positions of subparts can be manipulated by changing matrices at other levels of the tree. In this technique, the geometric properties of sub-objects (rotation, translation, scale, and so on) are determined as a function of these parameters in its parent object.

Implementing object positioning relationships with BOLIO's constraint system has two significant differences from the transformation hierarchy approach. First, relationships are described by a general graph, rather than a tree. This allows the state of an object to depend on the states of a collection of other objects. Second, relationships, in general, can be represented by arbitrarily complex programs. This allows the use of constraint information to selectively affect parameters of the dependent object. An example which makes use of these generalizations is the *link* relationship in BOLIO. This relationship, useful for modeling articulated figures, changes an object's rotation, translation, and scale so that it lies about a line connecting two points in space. A link relationship can be used, for example, to place a "forearm" object between "elbow" and "wrist" joints.

Other constraints currently available include **glue_points**, **glue_edges**, and **align_poly_normals**. The **glue_points** constraint translates a constrained object along a path which brings a specified vertex in contact with a specified point on another object. The **glue_edges** constraint calculates a rotation and scale for a constrained object such that a selected edge is coincident with an edge on another object. The **align_poly_normals** constraint sets the rotation of an object such that a normal with respect to the constrained object is parallel to a selected normal of another object. Combinations of these simple constraints can be used to provide high-level object placement control. For example, a user command of the form "put the cup on the table" could be implemented by constraining the up vector of the cup **bobject**'s **bearings** to be aligned with the negation of the table **bobject's** up, and by gluing a point on the bottom of the cup to a point on the top of the table. An extension of the standard file format is available for translating high-level keywords into specifications of object parts. An extended data representation is under development which will provide a standard method for associating natural language names with aspects of an object's structure.

Another constraint useful for constructing articulated figures is the inverse kinematics constraint. Inverse kinematics techniques have been used in several systems for the animation of human and animal figures[12,13]. This technique, adapted from robotics research, calculates the positions and orientations of joints in a limb based on movements of the ends of the limb (*base* and *end effector*). In BOLIO, objects are designated to represent the base, joints, and the end effector of the limb. When any object in the limb is moved, all other objects are repositioned in accordance with

the limb description. When the inverse kinematics constraint is initiated, a description of the limb is extracted from the positions of the objects selected as joints. This standard description of the limb is attached to the constraint structure and is subsequently available when the constraint is to be satisfied.

To add a new constraint to the system, the programmer provides a set of standard interface routines which are plugged into the constraint system. These functions include an **init** function which gets an argv/argc list from a command line constraint editor, a **print** function which prints the connections and the current state of the constraint to the standard output, and a **sat** (for "satisfy") function which updates the state of the dependent object based on its current state and the current states of the objects on which it depends. If the **sat** function changes the states of any objects, it informs the constraint system that the objects have changed. The constraint system then adds all the constraints dependent on those objects to the pending list -- checking to add a constraint only once.

The constraint system allows an application programmer to define an aggregate object dependent on a set of parameters derived from some set of other objects. As an example, consider how we could use constraints to build a simple Hermite patch display and manipulation routine. The Hermite patch is defined as a function of 16 objects: four control points, eight tangent vectors, and four twist vectors. We could represent each of the objects graphically — control points as small spheres, say, and tangent and twist vectors as arrows. These spheres and arrows would all be described by **bobjects** and could be manipulated using BOLIO's interface routines. All of these facilities are available to the applications programmer in a library of function calls. The Hermite patch code would derive its input directly from the **bearings** of the control points and the arrows. We would also need to constrain the ends of the tangent and twist vector arrows to remain fixed to the control points.

Now we can define a constraint which invokes routines to recompute the Hermite patch based on the current parameters whenever any of the 16 defining objects is moved. The patch would be displayed by computing the appropriate list of, say, move/draw commands and storing them in the **drobject** field of the patch's **bobject** structure. Now the user can graphically manipulate the patch and its parameters, and see the results displayed in one of BOLIOs viewports.

Using BOLIO objects as the parameters to higher-level data types has three main advantages. First, it allows sharing of the parameters between instances of the higher level type, so, for example, different Hermite patches can share an edge if both are dependent on common control point/tangent/twist assemblies for that edge. Thus when any of the objects for that edge are manipulated, both patches are recalculated. The second advantage is that the parameter objects can be manipulated as a result of other constraints or as the result of the behaviors of other objects in the environment. The positions of control points may be affected by dynamic simulations or other behavioral code — if, for example, an animated figure were to lift a corner of a piece of cloth, represented as a set of patches. (Witkin, et al[14], have discussed the use of energy constraints for computing mechanical and geometrical behaviors for models with many degrees of freedom). A final advantage is that the new application can take full advantage of BOLIO's interactive object placement facilities in the design of the editing interface for the new object.

The graph of relationships is implemented through an intermediate directional constraint structure. This structure contains a pointer to the object to be changed by this constraint, a list of pointers to the objects on which the constrained object depends, a code indicating the type of constraint, and a general pointer for keeping state information for the constraint. Each object which can be constrained keeps a list of pointers to the constraint structures which affect it, and a list of pointers to the constraint structures which depend on it. When an object's properties are modified (through user input, or through constraints), all the constraint structures for that object are added to a list of pending constraints. By iterating through the list of pending constraints, the constraint system traverses the relationship graph, and makes the system consistent with the defined constraints.

Since the constraints are represented in a general graph rather than a tree, problems can arise when cycles are present in the constraint network. One solution to this problem is to preprocess the network and allow only acyclic graphs. Acyclic graphs can be valuable in many simulation

situations where all relationships are strictly causal, and no information needs to be passed back through the network. However in many physical simulations, information needs to be passed back and forth. An example of this is the information passing from a shoulder muscle through the joints of an arm. If a collision is detected at the hand, the force of that collision must be passed back, where it will effect the path of the arm and possibly the rest of the body. Relaxation techniques are currently being explored to deal with this type of situation, as well as other more direct solution techniques.

5. Conclusion

We have described a graphical front end for viewing and manipulating objects as part of a 3D microworld. As such, BOLIO is an important component of an animation environment under development. The major elements of BOLIO include a set of largely device-independent structures for managing graphical objects and viewports, graphic i/o, and menus; a constraint package; and object editing facilities.

One application, *Pathtool,* a solid modeler based on generalized cylinders, has been incorporated, and we are in the process of integrating a figure animation system, *sa,* into BOLIO. *sa* has been described in detail elsewhere[15,16]; its main features include routines for describing and manipulating jointed figures, an event-driven simulation mechanism, and an animation language tuned for controlling jointed figure motion. The facilities we have described here will make it possible for us to simplify *sa* by relying on BOLIO's more general facilities for controlling linkages and graphic functions, and at the same time expand the capabilities of *sa* using the constraint and inverse kinematics routines BOLIO provides. In this way we hope to implement a software testbed for developing tools for modeling and controlling the behaviors of simulated agents.

Acknowledgments

BOLIO is a part of a large software environment to which many have contributed. Brian Croll and David Chen wrote early versions of graphic database management and display code. Pathtool was designed and implemented by Paul Dworkin. David Chen coded the inverse kinematics routines. And our special thanks to our many helpful colleagues at the Media Lab.

References

1. D. Zeltzer, "Towards an Integrated View of 3-D Computer Animation," *The Visual Computer*, vol. 1, no. 4, pp. 249-259, December 1985. Reprinted with revisions from Proc. Graphics Interface 85.
2. J. F. Blinn, "Systems Aspects of Computer Image Synthesis," *Course Notes, Seminar on Three Dimensional Computer Animation*, Boston, July 1982. ACM SIGGRAPH 82
3. F. C. Crow, "A More Flexible Image Generation Environment," *Computer Graphics*, vol. 16, no. 3, pp. 9-18, July 1982. Proc. ACM SIGGRAPH 82
4. T. Whitted and D. Weimer, "A Software Test-Bed for the Development of 3-D Raster Graphics Systems," *Computer Graphics*, vol. 15, no. 3, pp. 271-277, Dall, August 1981. Proc. ACM SIGGRAPH 81
5. T. Nadas and A. Fournier, "GRAPE: An Environment to Build Display Processes," *Computer Graphics*, vol. 21, no. 4, pp. 75-84, July 1987. Proc. ACM SIGGRAPH 87
6. M. Potmesil and E. M. Hoffert, "FRAMES: Software Tools for Modeling, Rendering and Animation of 3D Scenes," *Computer Graphics*, vol. 21, no. 4, pp. 85-94, July 1987. Proc. ACM SIGGRAPH 87
7. R. Parent, "A System for Generating Three-Dimensional Data for Computer Graphics," Ph.D. Thesis, The Ohio State University, Columbus, Ohio, 1977.
8. S. P. Ressler, "An Object Editor for a Real Time Animation Processor," *Proc. Graphics Interface 82*, pp. 221-225, Toronto, Ontario, May 1982.

9. J. Gomez, P. MacDougal, and D. Zeltzer, "A Tool Set for 3-D Computer Animation," *Course Notes, Introduction to Computer Animation*, Minneapolis, MN, July 24, 1984. ACM SIGGRAPH 84

10. C. L. Jackins and S. L. Tanimoto, "Oct-Trees and Their Use in Representing Three-Dimensional Objects," *IEEE Computer Graphics and Image Processing*, vol. 14, pp. 249-270, November 1980.

11. J. D. Foley and A. Van Dam, *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, 1982.

12. M. Girard and A.A. Maciejewski, "Computational Modeling for the Computer Animation of Legged Figures," *Computer Graphics*, vol. 19, no. 3, pp. 263-270, July 1985. Proc. ACM SIGGRAPH 85.

13. K. Sims and D. Zeltzer, "A Figure Editor and Gait Controller for Task Level Animation," MIT Media Lab, August 1987. Submitted for publication.

14. A. Witkin, K. Fleischer, and A. Barr, "Energy Constraints on Parameterized Models," *Computer Graphics*, vol. 21, no. 4, pp. 225-229, July 1987. Proc. ACM SIGGRAPH 87

15. D. Zeltzer, "Motor Control Techniques for Figure Animation," *IEEE Computer Graphics and Applications*, vol. 2, no. 9, pp. 53-59, November 1982.

16. D. Zeltzer, "Representation and Control of Three Dimensional Computer Animated Figures," Ph.D. Thesis, Dept. of Computer and Information Science, Ohio State University, August 1984.

A System for Algorithm Animation

Jon L. Bentley
Brian W. Kernighan

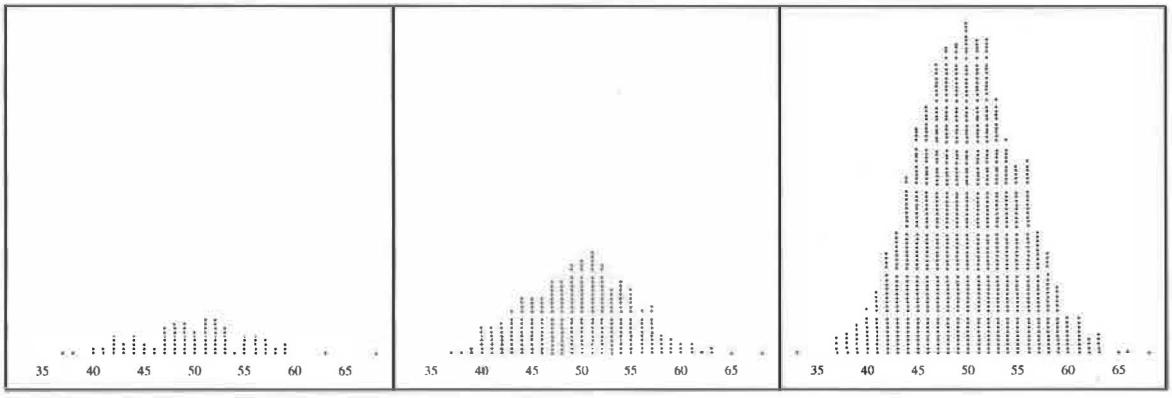
AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974

ABSTRACT

A program or an algorithm can be animated by a movie that graphically represents its dynamic execution. A sort, for instance, might be animated by a randomly scrambled sequence of lines being permuted into order. Such animations are useful for debugging programs, for developing new programs, and for communicating information about how programs work. This paper describes a basic system for algorithm animation: the output is crude, but the system is easy to use; novice users can animate a program in a couple of hours. The system currently produces movies on Teletype 5620 terminals and Sun and IRIS workstations; it also renders movies into "stills" that can be included in `troff` documents.

1. Introduction

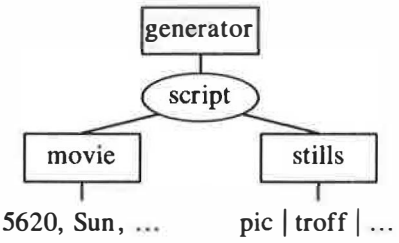
Dynamic displays are better than static displays for giving insight into the behavior of dynamic systems. Consider, for instance, the experiment of tossing a coin 100 times. The expected number of heads is 50, but the actual number obeys a binomial distribution. Probability theory tells us that the binomial histogram of counts will converge to the bell-shaped normal distribution; this sequence of pictures helps us appreciate the process more intuitively:



The three snapshots were taken after 100, 300, and 1000 experiments. Every tenth vertical dot is deleted to facilitate counting.

This paper describes the animation system that produced those pictures. A short program (a dozen lines of awk) performed the experiments and wrote the results to a *script* file describing the histogram's evolution through time. That file was processed by a program named `stills` to produce the pictures above, using `pic` and `troff`; we were able to control what frames were

displayed, in what size and form. A `movie` program displays the same data on a Teletype 5620 terminal or a Sun workstation; the viewer can control the speed of display, proceed forward or backward through time, and change the screen layout to emphasize certain views. These components can be depicted as:



Several systems have been developed for algorithm animation; see, for instance, "Techniques for Algorithm Animation" by Marc Brown and Robert Sedgewick in *IEEE Software*, January 1985, pp. 28-39, and the references therein. Most of those systems produce animations of very high quality; unfortunately, they are expensive in both programmer time and CPU time. Our system is at the opposite end of the spectrum: its output is primitive, but the system is easy to use; a new user can animate a simple program in an hour or two by adding a few lines of code. Although our system was designed primarily with program animation in mind, it can be useful in other domains as well.

Section 2 of this paper presents the details on the animation of one algorithm. Section 3 describes the system that produced the animation, and Section 4 shows other animations. Conclusions are offered in Section 5.

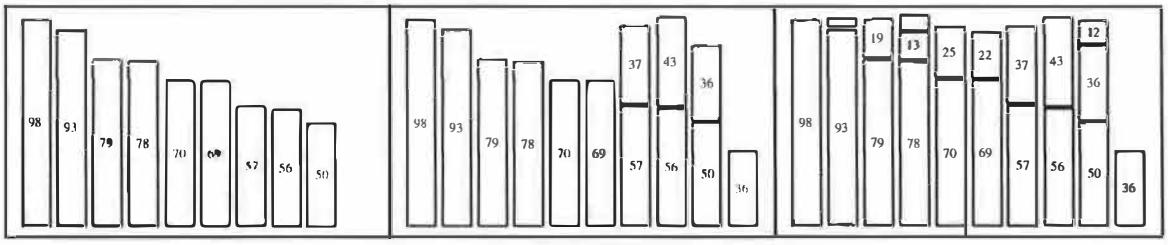
2. An Example — Bin Packing

The first part of this section uses animation to tell the story of an algorithm; the second part then describes how our system produced the animation.

The Algorithm.

Bin packing is a classical problem in computer science. The input is a set of weights between zero and one; we are to assign the weights to a minimal number of bins under the constraint that the sum of the weights in any bin is at most one. This problem arises in applications such as stock cutting and placing a set of files onto several floppy disks. Because the problem is NP-complete, researchers have investigated heuristics that give good, but not necessarily optimal, packings.

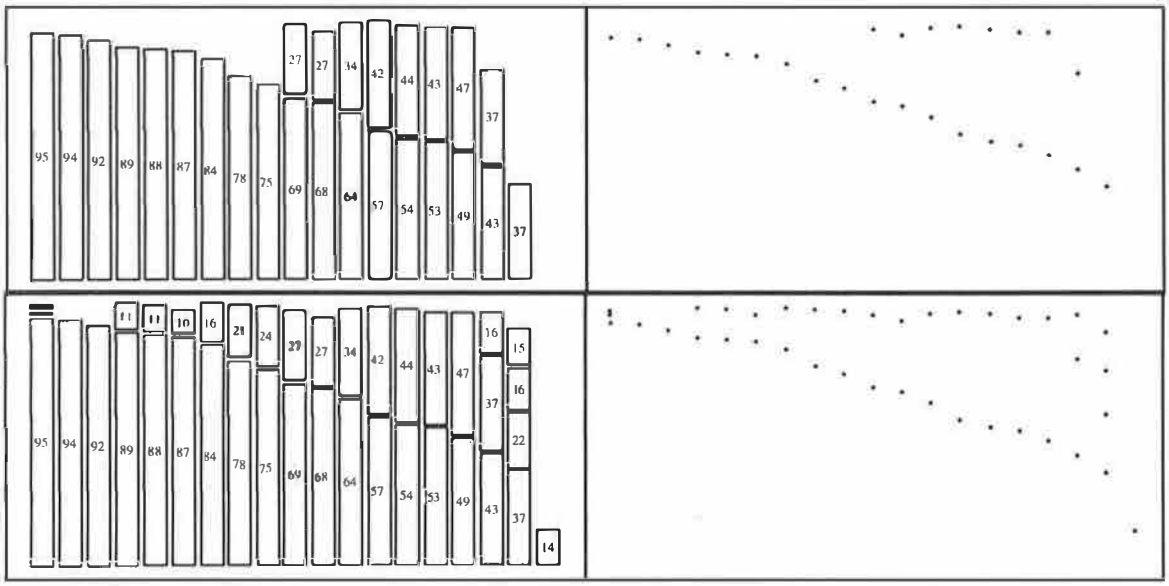
We will study the "First Fit Decreasing" or "FFD" heuristic (this heuristic is described in many algorithms texts). "Decreasing" means that the weights are considered from largest to smallest, and "First Fit" means that each weight is placed in the first (leftmost) bin in which it fits. This picture shows an FFD packing of 20 weights chosen uniformly from [0,1] (the numbers in each rectangle are the weights multiplied by 100 then rounded); the snapshots are taken after inserting 9, 13, and 20 weights:



weight: 9 weight: 13 weight: 20

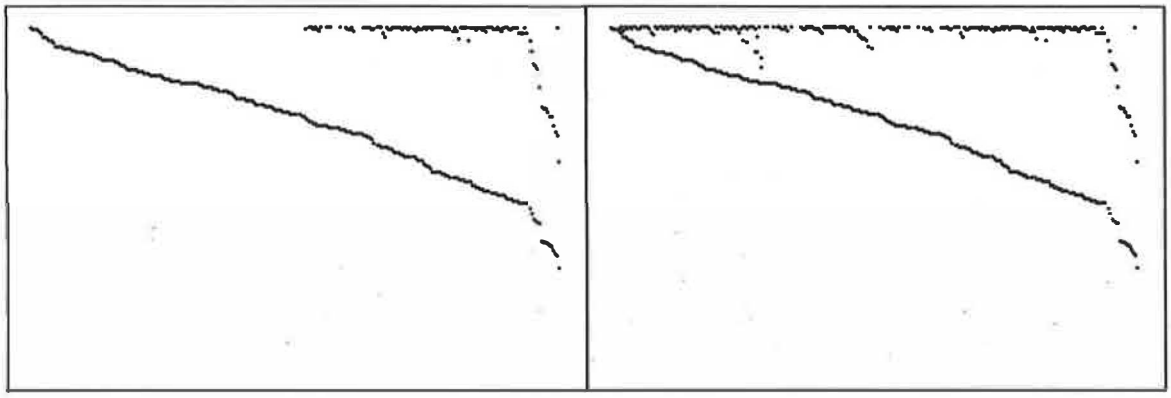
There are, of course, many ways to draw pictures of bin packings. Here are two side-by-side

views of packing 40 random weights; the top snapshot shows the packing after 26 weights have been inserted, and the bottom snapshot is the final state:



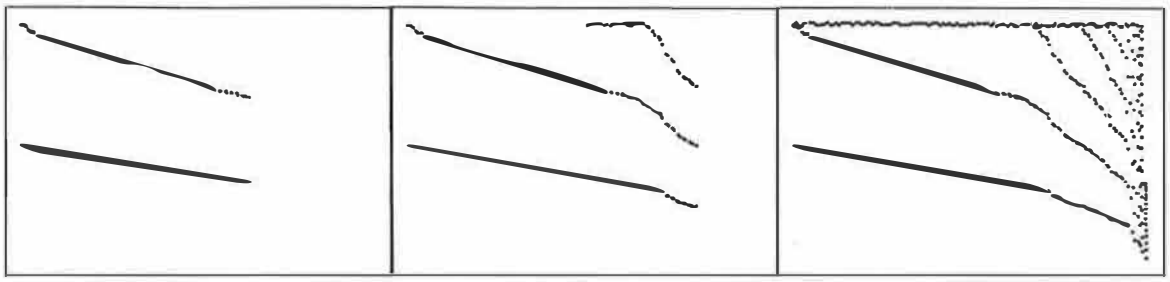
The right view is the old representation, in which weights are rendered as rectangles (with numbers in sufficiently large weights). It is fine for small instances, but cluttered for large packings. The right view places a dot at the top of each weight; it is an effective way to depict large packings.

The FFD heuristic produces very good packings when the weights are drawn uniformly over the range [0,1]. This picture shows 500 weights being packed, after 375 and 500 insertions.



The FFD heuristic essentially "folds" the weight list over on itself. There are a few large holes here and there, but, on the whole, the heuristic is quite effective.

When the weights are chosen uniformly from [0,.5], the FFD packings are even more efficient (the packings are optimal over 80% of the time). Here is the packing of 500 weights uniform on that range:



In the first snapshot, only weights greater than 1/3 have been inserted. The first and second weight go in bin 1, the third and fourth in bin 2, etc. The second snapshot shows the weights between 1/3 and 1/4: roughly a quarter of them "backfill" the old gap, while the remainder create a new "sawtooth" that will be backfilled by later weights. We won't give all the details, but the final snapshot shows that the resulting packing has a great deal of structure.

We have found pictures like these useful in several contexts.

Teaching. Movies are effective classroom tools, whether stored on videotape or controlled in real time by the instructor. Stills give less insight, but they allow longer contemplation and discussion, and are much more portable.

Programming. A simple bin packing program is very short and easy to get right; we'll see one shortly. Fast FFD programs, though, require a few hundred lines of very subtle code; pictures make the nightmarish task of debugging such programs fairly easy.

Research. Our interest in algorithm animation can be traced to the summer of 1983, when one of us (JLB) worked with Johnson, Leighton, and McGeoch on the mathematical analysis of the FFD heuristic. We spent roughly a programmer week writing a 5620 program to produce bin packing animations, and it was a wise investment: the pictures led us to several surprising conjectures and proofs.

The Animation.

The first step in using our system is to obtain a program to animate. Here is an `awk`† program that writes a history of a bin packing algorithm:

```
BEGIN {
    n = ARGV[1]; u = ARGV[2]; curmax = 1
    if (ARGC > 3) srand(ARGV[3])
    for (i = 1; i <= n; i++) {
        tw = u * (curmax *= exp(log(rand())/(n+1-i)))
        for (b = 1; bin[b] > 1-tw; b++)
            ;
        bin[b] = neww = (oldw=0+bin[b]) + tw
        print "insert weight" , tw, "into bin", i,
            "from", oldw, "to", neww
    }
}
```

All the action takes place within the `BEGIN` block (C programmers may think of it as `main()`). The first line sets from the command line the values of `n` (the number of weights) and `u` (the weights are distributed over the range [0, `u`]). The `for` loop packs each of the `n` weights; the first line in the loop body is probabilistic magic to ensure that the weights are uniform and appear in decreasing order. The inner `for` does a sequential search for the first bin in which a weight fits; the next two statements insert the weight and write a record of the insertion. We implemented the

† This program and `awk` programs later in this paper use features of the mid-1985 `awk` release described by Aho, Kernighan and Weinberger in *The Awk Programming Language* published by Addison-Wesley in 1987.

program in awk to make it concise.

To animate the program we replace the single print statement with several print statements:

```
BEGIN {
    n = ARGV[1]; u = ARGV[2]; curmax = 1
    if (ARGC > 3) srand(ARGV[3])
    print "#ffd_bin_packing n=" n " u=" u
    print "view dot\ntext 1 0"
    for (i = 1; i <= n; i++) {
        tw = u * (curmax *= exp(log(rand())/(n+1-i)))
        for (b = 1; bin[b] > 1-tw; b++)
            ;
        bin[b] = neww = (oldw=0+bin[b]) + tw
        print "view dot\ntext", b, neww, "dot"
        print "view rect\nbox", b-0.4, oldw+.01, b+0.4, neww
        if (tw > .1) print "text small", b, oldw+tw/2, int(100*tw+.5)
        print "click weight"
    }
}
```

To animate a packing of four weights chosen uniformly over the range [0, 1] we invoke the program with this shell command:

```
awk -f ffd.gen 4 1
```

That produces as output this "script file", printed in two columns to save space:

```
#ffd_bin_packing n=4 u=1
view dot
text 1 0
view dot
text 1 0.968228 dot
view rect
box 0.6 0.01 1.4 0.968228
text small 1 0.484114 97
click weight
view dot
text 2 0.388697 dot
view rect
box 1.6 0.01 2.4 0.388697
text small 2 0.194348 39
click weight
view dot
text 2 0.733216 dot
view rect
box 1.6 0.398697 2.4 0.733216
text small 2 0.560956 34
click weight
view dot
text 3 0.307457 dot
view rect
box 2.6 0.01 3.4 0.307457
text small 3 0.153728 31
click weight
```

This script file uses four commands: box, text, view and click.

A rectangle with opposing corners at (x_1,y_1) and (x_2,y_2) is drawn by a command of the form

optional_label: box x_1 y_1 x_2 y_2

(Literals are shown in typewriter font and categories are in *italics*.) Text is produced at (x,y) by a command of the form

optional_label: text x y *anything at all*

Coordinates can lie in any range; later programs will scale them appropriately.

The view command is used to place output in a particular view. There are two views here, dot and rect. Interesting events are marked by the click command. stills and movie can refer to each click with this mechanism. Labels, view names and click names are arbitrary and unrelated to one another.

The movie program is currently implemented on the Teletype 5620 terminal and the Sun workstation. The first step of the program is to read the script file once (typically from disk) and load it into local memory; during this process, the movie is played once from beginning to end. Subsequently, the viewer can examine it in greater detail: stopping and starting, backwards and forwards, faster and slower, etc. We will now sketch how to control these options.

Mouse button 1 is used for "stop" and "go". Button 3 does most of the work. Selecting

"again" repeats the movie. The speed is controlled by either doubling ("slower") or halving ("faster") the wait time at certain key events (the "clicks" mentioned above). This applies only in "run" mode; if one selects "1-step" mode in button 3, then each hit of button 1 moves to the next appropriate click. "Backwards" and "forwards" change the direction of play; together with "1-step", they make it easy to locate a key event in the movie.

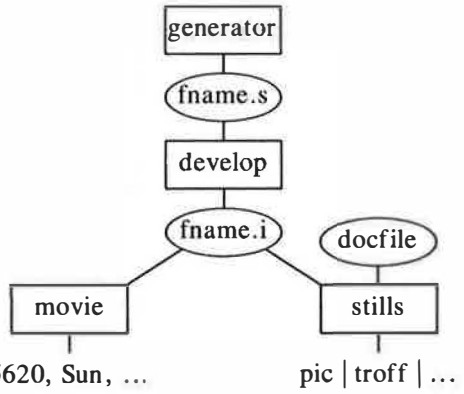
Button 2 lists the names of the views and clicks in the animation. When a view name is selected, you can sweep a rectangle in which that view is to be displayed; one can delete a view by sweeping its rectangle out of the window. Selecting a click name turns it on or off (the ones that are on have an asterisk next to the name). Clicks that are on cause a wait in run mode and a pause in 1-step mode.

In this system, the computation cannot be interactive (i.e., you cannot type in a number and watch a binary search try to locate it in an array). The *display* of a fixed computation is, however, highly interactive: the viewer can run it forwards or backwards, quickly or slowly, etc.

This system would have been very useful for the experimental bin packing research we sketched earlier. Several years ago, we had to build a special-purpose animation program on the Teletype 5620 in several hundred lines of C written in a week; we can now do the job in a dozen lines of code written in an hour. Our new system provides several useful facilities that were not present in the original program but which would have been very useful for our research: multiple views, stills output, and more control over presentation (backwards and forwards, one-step, etc.).

3. The System

This section gives a more detailed view of our animation system. A script file is processed by the heretofore unmentioned program named `develop`. The output of `develop` is an intermediate file that feeds `stills` and `movie`:



The Script and Intermediate Languages.

The script language is summarized in this table:

```
# comment
optional_label: line  options  x1 y1 x2 y2
        [-]  ->  <-  <->
        [solid]  fat  fatfat  dotted  dashed
optional_label: text  options  x y  string
        [center]  ljust  rjust  above  below
        small  [medium]  big  bigbig
optional_label: box  options  xmin ymin xmax ymax
        [nofill]  fill
optional_label: circle  options  x y  radius
        [nofill]  fill
view  name
click  optional_name
erase  label
clear
```

A line whose first non-blank character is # is a comment; blank lines are ignored.

If a label is present on a geometric object, it names the object and implicitly erases any existing object with the same name in the same view. Options for an object are indented on a subsequent line in this description, with defaults in brackets. The options are a (possibly null) list of names, terminated by the next numeric field. For instance, a script file might contain this command

```
a117: line <-> 0 234.021 1 234.087
```

to draw a line with arrows at both ends; it will have the default width **solid**.

The **view** statement places subsequent objects in a new view, and **click** denotes an interesting event.

A labeled geometric object can be explicitly erased by the command

```
erase  label
```

The various views have distinct name spaces; the same label may be applied to two unrelated objects in two different views. All objects in the current view can be erased by the statement

```
clear
```

The intermediate language can be viewed as the "assembly code" output of the **develop** program (which is about 1000 lines of C). The program scales all numeric values into the range 0..9999, translates symbolic labels into numbers, makes implicit deletes explicit, and translates options into a standard form. The resulting file is easy for the subsequent **movie** and **stills** to process.

The Movie Programs.

The original **movie** program runs on the Teletype 5620 and contains roughly 1500 lines of C. Movie production, as with most 5620 programs, uses a host process and a terminal process. The host sends the intermediate file produced by **develop** in a compact form to the terminal, which stores it in a form suited for forward or backward display.

The buttons were sketched in Section 2. In general, drawing can be interrupted at any point by pushing any button, then resumed by pushing button 1.

Four menu items control two variables in button 3: **faster** and **slower** decrease (halve) and increase (double) the pause at selected clicks, and **thinner** and **fatter** alter the width of lines. Three menu items control binary attributes:

```
backward      forward
or mode       xor mode
1 step        run
```

The `new file` selection allows the viewer to read a new intermediate file (without downloading the program again), and `exit` leaves the program.

Button 2 lists views and clicks. Selecting a view results in an icon for sweeping a rectangle. Views may be positioned anywhere; portions positioned outside the window will not be shown. Initially, views have a 5 percent margin at each edge; this margin is zero for views that have been reshaped.

As it is for the 5620, so it is for the Sun workstation, although the exigencies of the Sun window system have forced us to curtail some features. To keep the code relatively portable, there are again two processes, so the window in which one starts the animation clones another window of uncontrolled size, shape and position where the animation itself occurs.

The `movie` programs on both the 5620 and the Sun are designed for interactive exploration of computations. We have also implemented a version of `movie` on the IRIS workstation aimed at producing videotapes of a quality suitable for classroom use. In some ways it is less powerful: it runs only in the forward direction and does not have single stepping. In other ways it is more powerful: the viewer has greater control over the positioning of views and the time spent pausing at clicks, and we have added eight colors as options on any geometric object. In any case, the programs are different: the original `movie` is controlled by a mouse, while the IRIS version has textual input.

This `movie` program shows that the animation system is easy to port. With no previous experience on the IRIS, we had the first version of the program up and running in one day and 250 lines of C. The final program is 500 lines of C, and was finished in three working days.

The Stills Program.

The `stills` program is a typical `troff` preprocessor. Portions of its input bracketed by `.begin stills` and `.end` are translated into `pic` commands, and the rest of the input is passed through untouched. A paper containing `stills` input is typically compiled by a command like

```
stills paper | pic | troff >paper.out
```

For instance, the second bin packing picture in Section 2 was produced by this description:

```
.begin stills
file ffd2.s
view rect ""
view dot ""
print weight 26 40
frameht 1.5
framewid 3.0
down
times invis
small -5
.end
```

The first line names the script file, and the next two lines select views for display and give them null titles. The `print` statement causes snapshots at the selected times of the click `weight`. The five remaining lines are name-value pairs: the height and width are in inches, `down` causes time not to go `across` the page (that name allows the null value), the click times are not displayed (`invis`), and `small` text is rendered five points smaller than usual.

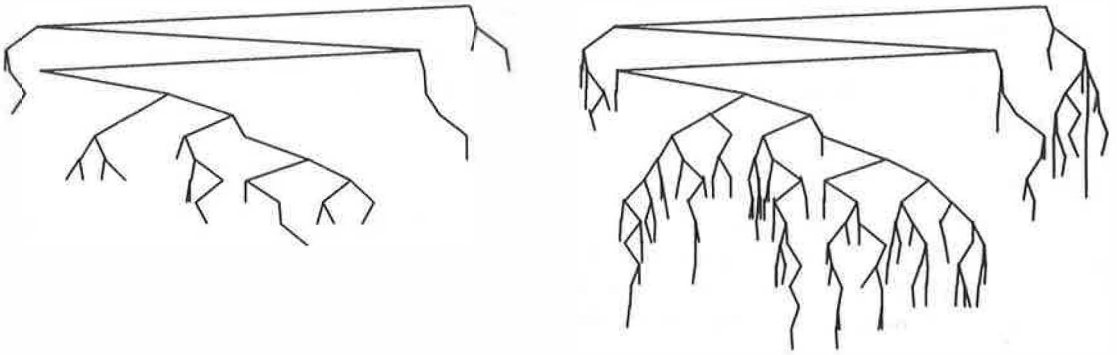
In summary, `stills` input consists of these commands:

```
print all
print final
print clickname all
print clickname number number number ...
view name optional title
parameter_name value
```

At least one `print` statement and a `file` assignment are mandatory; other statements are optional.

4. Using The System

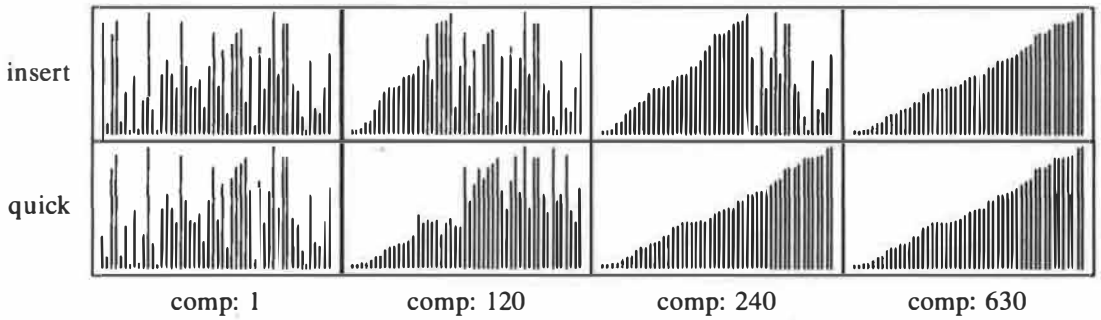
Our system provides only a few geometric primitives: lines, boxes, circles and text. Nevertheless, they appear to be sufficient for animating interesting algorithms. Here, for instance, is a binary search tree after inserting 50 and 200 random elements:



The script file was produced by a 23-line `awk` program and uses only lines; the `stills` description is 6 lines long.

The system that we have described is the bare bones of an animation environment. In the spirit of UNIX, we have enhanced the environment not by modifying the primary programs, but rather by using small filters that interact with the various files in the system.

Here, for instance, is a "race" of insertion sort against quick sort on an array of 50 random numbers:



Quick sort finishes after 240 comparisons, while insertion sort requires 630 comparisons. A 32-line `awk` program generated both sorts; we used an editor to change the call in the main procedure. (Some sophisticated animation systems require 500 lines of code to produce beautiful animations of insertion sort. Our output is unpolished by comparison, but it is adequate for many purposes.) While other algorithm animation systems implement races with a general mechanism for time sharing, we did the job with a dozen-line `awk` program that merges two files.

We have built several useful filters in addition to `merge`. The program `show.clicks` takes

a script file as input; its output is a new script file containing all information in the input and, in addition, a new view named `click.count` in which the various clicks are counted. This is useful for preparing `stills` files and for debugging.

Another program processes lines in the script file of the form

```
#var name value
```

The output script file has a new view named `variables`; it contains the name of each variable mentioned and its current value. The program `view.clicks` prints a summary of the views and clicks used in a script file; it is helpful as one is preparing a `stills` file. The system does not have a facility for counting clicks; rather, we use filters such as

```
grep 'click comps' | wc
```

to see how many comparisons were made. We will even admit to using text editors to make minor changes to both script and intermediate files.

Larger filters have also proven useful. For instance, we built a set of tools to render animations of three-dimensional lines and text (circles and rectangles were not supported). The primary program translated a three-dimensional script into a standard script that contained two two-dimensional views for each three-dimensional view; the resulting `movie` and `stills` were suitable for viewing with standard stereo viewers. Support programs included a filter for rotating a view around a given line.

5. Conclusions

The examples in this paper illustrate the capabilities and limitations of our animation system. The output of `movie` is a crude but useful animation. The output of `stills` is handy for more detailed study and for presentation in documents (we would like to include a movie in this document, for instance, but paper is easier to distribute than videotape).

If our system is so crude, why bother using it? Why not animate an algorithm simply by drawing geometric objects on the output device you happen to be using? Some of the answer lies in services like these:

Device Independence. A script file can be viewed interactively as a movie on a 5620 or a Sun; a videotape can be made on the IRIS. The same script file can be incorporated into a document by `stills`. The system is easy to port to additional output devices.

Names. Labels allow geometric objects to be erased; implicit erasure by re-using a label avoids much of the tedium of bookkeeping. Click names mark key events; they can be used to group related events.

Independent Views. Different simultaneous views of a process are crucial for animating algorithms. In our system, a single statement moves from one view to another. Within a view, the user need not be concerned about the range of coordinates; the system scales automatically.

Viewer Control. Both `movie` and `stills` allow the viewer to select which views will be displayed and which clicks will be recognized. Additionally, `movie` allows the viewer to go forward or backward, in single steps or running at a selected speed.

An Interface To The World. Although writing to files takes more computer time than using the geometric primitives provided by a specific output device, those files allow complicated tasks to be easily composed out of simple software tools.

Our system does not support interactive animations, however: once the script has been generated, there's no way to change it except to generate it again.

Acknowledgements

We are deeply indebted to Howard Trickey; he gave us invaluable advice for getting a minimal animation facility working in the Sun environment, then finished the job properly. Andrew Hume and Jane Elliott made possible our first experiments with animation. Our early users, Rick Becker and Chris Van Wyk, gave us bug reports and suggestions for improvements.

Distributed Computation for Computer Animation

John W. Peterson
Computer Science Deptartment
University of Utah

Abstract

Computer animation is a very computationally intensive task. Recent developments in image synthesis, such as shadows, reflections and motion blur enhance the quality of computer animation, but also dramatically increase the amount of CPU time needed to do it. Fortunately, the computations involved with computer animation are easily decomposed into smaller tasks, such as rendering single frames or parts of a frame. This makes the problem an ideal candidate for "coarse-grain" parallel implementation.

In order to provide the necessary cycles, unused idle time on personal workstations is used to provide a single large parallel computing resource. A survey of several schemes for coordinating this type of resource is presented, along with a detailed examination of a Unix based system currently in use at the University of Utah.

1 Introduction

As large networks of computers become commonplace, it has become interesting to consider them as a single computational resource, rather than individual machines. Recent advances in networking software such as distributed filesystems and remote procedure calls make using networks much more transparent to application programs. For some applications, the ability to use multiple machines in parallel is limited (e.g., complex simulations where the next iteration depends on data from the previous one). Other applications, such as computer animation, are ideally suited to parallel execution. This paper examines this type of application on large networks.

The type of parallelism explored in this paper is assumed to be course grained, with individual computations lasting minutes or hours instead of fractions of a second. Another assumption is that the computing resources are developed and maintained for general purpose use. In other words, we wish to take advantage of an existing resource, rather than develop one specifically for the task.

This paper gives a brief discussion of the scale of the resources we are discussing, and then presents an informal survey of existing systems for using computational power on networks as a whole. Finally, a system developed at the University of Utah for computer animation is examined in detail.

2 The Resources

The resources available on a network of workstations are dependent on two factors: how much the workstations are used by their dedicated users, and the power of the individual machines.

2.1 CPU usage

In addition to the obvious periods of idle time (night, weekends, etc.) a typical workstation CPU is usually not fully utilized even during the day. A workstation often spends its time performing relatively simple tasks, such as editing, reading mail and terminal emulation. Statistics gathered indicate a typical workstation CPU spends approximately 90–95% of its time in the idle loop[1].

Unfortunately, not all this idle time is directly available. If the CPU's idle loop is replaced with a major application, distracting side effects occur even if the application is running at a low priority. For example, if the workstation user is interacting with a Lisp interpreter, having another large application in the background may dramatically increase the paging activity and slow down the interaction.

2.2 CPU power

The availability of advanced microprocessors like the 68020/68881 chip set have blurred the distinction between mainframe and workstation computing power. Some recent benchmarks conducted at BRL [6] give the approximate comparisons:

68020 based workstation $\approx$ 1 Vax 780

4 processor Cray XMP/48 $\approx$ 90 Vax 780's.

So if you can get efficient parallelism:

90 workstations $\approx$ 4 processor Cray XMP/48

The economics of this comparison are interesting, since a Cray goes for something like \$10–\$15 Million vs. \$3-\$5 Million for a large workstation network.

3 Getting Parallelism The Hard Way

There are many examples of animation done by manually starting the computation on a number of machines. Among the best known are:

- Jim Blinn's animation of DNA molecules for the PBS *Cosmos* series. Blinn and his collegues wandered all over NASA's Jet Propulsion Laboratory after 5:00 pm looking for unused PDP-11's. When one was found, a tape was loaded on the machine and it was left to crunch away on part of the sequence for the evening. The results were collected on magnetic tape in the morning.[10]

- The short film *The Adventures of Andre and Wally Bee* produced by Lucasfilm was done on a larger geographic scale. Portions of the film were computed on a Cray in Minnesota and on ten Vaxes at MIT's project Athena. Data was shipped out and results were collected on tape. The final results were composited at Lucasfilm's facilities in California.

[1] Usage statistics were taken on the Sun and Apollo workstations in the CS department. The HP workstations don't appear to keep track of this information.

- Apollo Computer's film *Quest - A Long Ray's Journey Into Light* was computed on a few hundred workstations at Apollo. Although the machines were connected with a local area network, in the rush to complete the film little software was written for coordinating the computation. Instead, a person (given the screen credit 'Node Hunter and Gigabyte Master') typed the necessary commands into individual nodes. Since that project, more advanced software has been developed for starting the computations [1].[2]

4 Systems for Distributed Computation

4.1 The Xerox "Worm" Programs

One of the earliest examples of a system for performing distributed computation on a local area network is the Worm developed at Xerox PARC[9]. The worm worked as a layer on top of which applications were built. The program was executed on several machines at a time, each machine a *segment* of the worm. The worm worked at a relatively low level compared to more modern systems.

After a worm was initially started, it operated by attempting to fill out the rest of its segments. It would work through the network incrementally, probing nodes to find out if they were idle. When an idle node was found, the worm would continue to boot itself on the idle machines until it had filled out its segments. The segments of the worm communicated with each other using a limited broadcast, or *multicast* protocol.

Because the worm operated at such a low level (there was very little operating system support beneath it) controlling it was a major problem. If the worm encountered a serious problem it could crash the workstation it was running on. If a worm became corrupted as it moved from machine to machine, the corrupted segment might run, but would spawn new segments that would crash. Since the original worm thought it needed to fill out the rest of its segments, it would continue trying to boot until all of the machines on the net had crashed (the paper describes a situation where this actually happened).

One of the applications for the worm described by the paper is a multi-machine animation system. The worm was modified so one machine could serve as central control node. This in turn spawned a series of smaller worms that located the worker nodes. The master node (itself not part of a worm) would send out the basic scene description to the worker nodes, and would later collect the results.

4.2 Recent distributed computation systems

The Xerox Process Server Recently, Xerox has developed a system knows as the Process Server [4] for workstations running their Cedar environment. This system is designed to make excess cycles on a workstation available to other users on the net in a relatively transparent fashion. The system uses remote procedure calls and transparent access to file servers for communication between nodes. Three types of entities are provided by the system: *Clients*, the workstations that request services; *Servers*, the machines allowing computations to be run on them; and a *Controller* which processes the client's request and assigns a server to it.

[2] The films *Quest* and *The Adventures of Andre and Wally B.* appear in the anthology *Animation Celebration*, released by Expanded Entertainment in 1986.

When a user makes a request for work, the parameters (commands, arguments, etc.) are passed to the Client process on the user's workstation. The Client then contacts the Controller, and if the request seems valid, the Controller selects a server machine (based on which one appears least loaded) and returns the machine's identifier to the Client. The Client then contacts the Server. The Server fetches the files it needs, and starts executing the command. During execution, the Server uses the Client for file operations and answering questions about the user's environment. If an error occurs, the Server brings up an error window on the Client's node. If the Server aborts the computation (because the load was too high) or crashes, the Client must ask the Controller to assign it a new Server and restart the computation.

The system is implemented using a Remote Procedure Call (RPC) protocol. It is designed to be relatively transparent to the applications executed by it, with few changes needed to the source. It is intended for relatively large granularity computing (compilation, typesetting, image generation, etc.). Because the system uses specific, lightweight protocols, the Process Server runs with relatively little overhead.

Apollo's Network Computing System Apollo Computer recently developed a system called the Network Computing System (NCS) for sharing resources (including computation) across large networks of heterogeneous machines[3]. It provides a Remote Procedure Call interface, a network data representation definition, an interface compiler and support for replicated databases. The remote procedure call interface supports several scalar formats. These are automatically converted for different hosts (to compensate for differences in floating point, byte ordering, etc.). A Network Interface Definition Language automatically constructs the RPC networking interface from user-defined stubs. It also provides methods for passing more complex data structures over the network, such as trees.

NCS provides a Location Broker, a service that allows objects on the network to be found by type, interface or combinations of these characteristics. They are identified by Universal Unique Identifiers that are guaranteed to be unique across the network. Although NCS supports remote computation, it currently doesn't provide for automatically selecting hosts for the remote computation on the basis of load. This is planned as a future extension; nodes will be able to query a "compute slot allocator" to access a replicated database of candidate nodes.

Remote Unix Remote Unix (RU) is a system developed by Michael Litzkow at the University of Wisconsin [5]. This system is designed to allow a single process to operate for a very long time, migrating from machine to machine as various workstations are used or become idle. A unique feature of RU is that processes can be completely checkpointed as they execute, including the status of open files. This takes place when a user logs into a workstation. When the RU spooler finds another machine to restart the computation on, it resumes the checkpointed computation without loss of work. This facility also gives RU a large degree of fault tolerance, since if a machine crashes the process can always be restarted from the last checkpoint file.

The control system contains two components, a central *resource manager* for gathering information about all the available machines, and a *local scheduler* to make decisions affecting a particular workstation. The resource manager periodically polls the schedulers to determine which workstations are accepting RU jobs and what jobs are waiting to run. When it finds an "idle" workstation, it sends a message to the waiting job granting permission to execute on the idle machine.

RU has been in use at Wisconsin on a large network consisting of several larger Vaxes (11/750's, 11/780's) and about 100 MicroVax workstations. In one case a single job was able to accumulate 60 CPU days over a three month period. Although the system supports parallel execution by queuing several jobs at the same time, no statistics have been gathered on this mode of operation.

5 Some Example Systems

In this section we present some examples of systems actually used for animation or similar purposes. Because these systems are usually built around existing environments informally, there are not many published examples of them. In order to provide more examples, a poll was conducted on the Usenet and Arpanet networks requesting information about these types of systems. Most of these examples are from this poll.

(A note about notation: In the descriptions below, the word *dispatcher* means the machine responsible for controlling the computation. The computations are executed on *worker* nodes.)

5.1 Apollo/MBX based system

Part of the animation system described in [7] contained a method for using a large number of Apollo workstations. It was based on Apollo's MBX ("Mailbox") system routines. These routines allow inter-process communication between multiple workstations via filesystem objects known as mailboxes. After the dispatcher process opens a mailbox, the workers can open connections to the dispatcher process via this mailbox. Since all the Apollos on the network share the same filesystem, they can all open connections to this mailbox.

This system required the ray tracer to be modified to call the routines:

Init Opened the inital connection to the dispatcher's mailbox.

Send_status Sends a progress report (e.g., the current scanline number) to the dispatcher.

Test_login Asked the node if anybody had logged into it. If this routine returned true (somebody had logged in) the program is expected to call the next routine:

Shutdown Informs the dispatcher this node is no longer available, closes the MBX connection, and exits the program.

Finished Informs the dispatcher this node is finished with its task and can start on another.

The dispatcher would first open the MBX file, and start the worker processes on the remote nodes. A simple protocol was used for giving each worker a unique ID to identify itself, since the dispatcher received all of its input on a single channel. The dispatcher listened to messages generated by Send_status, Shutdown and Finished, and updates its record of the work done. If Finished was called, that workitem would be removed from the queue, if Shutdown was called it would be re-queued. Every transaction was recorded in a logfile.

As implemented, the system could tolerate worker failures but not dispatcher failures. Although never implemented, some ideas were planned for increasing the robustness. This included assigning one of the workers to be the "copilot". The copilot would receive a copy of the

dispatcher's state every time it performed a Send_status call. If the copilot tried to perform a Send_status and failed (i.e, the MBX channel was no longer open) it would spawn off a local copy of the dispatcher. This new dispatcher would re-open the MBX channel. If other workers tried to perform a Send_status and failed, they could try to close and re-open the MBX file to establish a connection to the new dispatcher.

Although the basic system was used to generate a few stills, it was never used for large scale work, mainly because the bulk of the computing resources were on other (non-Apollo) systems. The system was dropped, eventually replaced with one that could take advantage of any Unix host.

5.2 Locus based system at UCLA

At UCLA, Matthew Merzbacher developed a scheme for generating frames with a collection of ten Vaxes running the Locus operating system. Locus provides a common shared filesystem across several machines in a Unix environment. This allowed all of the Vaxes to access a single directory where all of the data and results were kept. The files were named after the worker machines and given suffixes indicating the state of the system (e.g.: athena.i, athena.r, athena.d).

After the dispatcher started, it created **.i** files named for each of the worker machines, containing the data for the rendering program, and spawned rendering processes on all of the workers. The worker process polled the main directory, waiting for a **.i** file with its name on it. When one was found, the worker created the **.r** file to indicate it was running. When it finished the job, the worker removed the **.r** file and created a **.d** file, indicating it was done.

The dispatcher polled the directory looking for the **.d** files. When one was found it removed the **.i** and **.d** files (in that order, to prevent the job from running twice) and places a new **.i** file in the directory. If the load on a machine was too high or if it was past 7 am, no new jobs would be started. Logs were kept of how many jobs per night were completed. Each job corresponded to a frame of the movie and required approximately ten minutes of Vax 11/750 time.

If the dispatcher failed, there was a backup dispatcher waiting to take over (which in turn would spawn a new backup). A new dispatcher was automatically started each evening with the Unix *at* utility. It could detect if a job had failed the previous night, because the directory would contain a **.i** file without a corresponding **.d** file.

5.3 TCP based system at BRL

At the Army Ballistics Research Laboratory, Mike Muuss developed a system for taking advantage of idle time on large mainframes and supercomputers. The ray tracing program was modified so it could operate remotely, receiving and writing information over a TCP connection. The work is dispatched to the worker machines in small portions of a frame (e.g., three scanlines) and collected by the dispatcher after each scanline is finished. Each worker has a private copy of the database, but the information specific to the frame being rendered (viewpoint, positions of objects, etc.) is transmitted directly to the worker via point-to-point TCP connections. Machines are selected manually, and can be added and dropped from the pool of workers on the fly. The rendering process runs at a very low priority on the worker mainframes.

If a worker fails, the job running on it is automatically re-queued on the next available machine. However, the dispatcher writes out the data collected from the workers after every

frame, so the entire frame is lost if it fails. The system is usually run with mainframes or supercomputers, in one instance 13 Gould 9000 series machines "all over the east coast" were used.

5.4 Lisp Machine based system at the MIT Media Lab

At the MIT Media Lab, Steve Strassmann developed a system for using up idle time on Lisp Machines. When each host boots, it creates a copy of the idle time "server" daemon. This daemon remains dormant until it detects the machine is idle. Then the daemon wakes up and reads a job specification file from a central host. It picks a job to run and executes it. Synchronization and the division of labor are specified by the application the daemon is executing, not by the daemon itself.

If a job is interrupted by an error, it quits and the daemon goes back to the central job specification file to see what to do next. An arbitrary "clean up" procedure can be associated with a job, and is executed whenever the job exits (write to a log file, etc.). The job specification also allows for a maximum execution time for a job (kill it after N minutes of CPU time), uniqueness (only one copy of the job is run at a time) or logging (start and stop times are logged in a central file). If the central job description file is unavailable, the daemon goes to sleep until it can re-open it.

The system is not tied to any particular task. Applications have included running diagnostics on a connection machine, and 'frivolous console animations'.[3]

5.5 File based system at NYIT

While at NYIT, Paul Heckbert developed a scheme for soaking up the idle time on eight Vaxes there. Because the systems went down at least once a day for backups and loads across machines were uneven, the system had to be fault tolerant, de-centralized and able to deal with loaded and unloaded machines.

The boot script for each of the vaxes was modified to start a daemon responsible for running the computation. This daemon would read a "job/log" file containing a list of shell commands to run and the status of each. For example, a job/log file for computing five frames of animation might look like this:

```
done(vaxb, vaxg)        gen.sh 0
done(vaxc)              gen.sh 1
done(vaxa)              gen.sh 2
running(vaxa)           gen.sh 3
-                       gen.sh 4
```

This file means that frames zero through two are done, frame zero was computed on two machines (vaxb and vaxg), frame three is being computed on vaxa, and frame four hasn't started yet.

The daemon read this file and picked a job to run based on the following priorities:

[3]This is much like the applications for the Xerox PARC "Worm" program.

1. If a job is listed as running on the machine reading the file, then it must have crashed, so resume work on that job. The ray tracing program was written so it could resume computation in mid-job to minimize lost work.

2. Run an unstarted job, if any are left.

3. Run a running job. This is useful in the case of a another machine crashing or slowing down due to a heavy load.

Each machine decides which jobs to run, there is no single master machine. Just before a machine started up a job, it would update the status in the job/log file and then copy it over the network to the other machines in the pool. The shell script started by the daemon saved its output in a common directory. This was inspected once a day and the results transferred to a big disk.

The system was used for three large jobs:

- An animated sequence of 120 ray-traced frames. It took 84 CPU-days over a period of 19 days on seven Vaxes (six Vax 780's and one Vax 750);

- An eight CPU-day ray-traced image of a morphine molecule (computed at a resolution of 2048x2048);

- Computing all amicable number chains up to 200,000,000 (a number theory problem).

On the larger jobs, the system was able to use 65% to 75% of the available CPU time. It was able to recover from machine crashes and shutdowns, and ran around the clock on seven machines for several weeks. The only significant problem was the job/log files becoming inconsistent on the various machines, probably because there was no locking scheme for the job/log files.

5.6 Systems at Xerox PARC

While at Xerox PARC Steve Schiller developed a system for using approximately 100 workstations to compute an animated sequence of fractal images. In that system, one of the workstations served as the main dispatcher for the computation. It made a remote procedure call to a worker machine, giving it the parameters for computing a particular frame. When the worker finished computing the frame it would inform the dispatcher that it was finished. It was up to the dispatcher to actually retrieve the frame. The dispatcher could also ask a worker if it was busy, and if so, when it expected to finish the frame. If somebody logged into the machine, the computation stopped, and the work was re-queued on the next available machine (code was implemented to re-start partial frames, but became a source of trouble and was dropped). The computation times ranged from five minutes to one hour per frame, depending on the complexity of an image (twenty minutes was the average).

The scheduling of the computations was complicated by disk space constraints on the dispatcher. The frames had to be recorded by the camera in the correct order and then removed to make room for new frames. However, the workers might not have finished them in the proper sequence. Since the dispatcher had only twenty frames worth of available disk space, there would occasionally be times when the dispatcher could not retrieve a frame because it didn't

have enough disk space. (Of course, enough space must be available for the frame the camera is waiting on).

In order to help avoid this problem, the dispatcher kept track of all of its outstanding frame requests and when they were made. When it wasn't busy with anything else it checked the machine working on the frame the camera was waiting on. If that machine was unreachable (crashed, net problems, somebody logged into it, etc.) or was taking a suspiciously long time, then the dispatcher re-assigned the frame to the next free machine. A log was kept of when each frame was retrieved, how long it took to complete it and the name of the machine that worked on it. This was useful for pinpointing slow or unreliable workers.

Once the kinks were worked out, the system was fairly reliable. Schiller estimates about nine out of ten 48 hour runs were without incident. The system achieved approximately 80% parallelism during operation.

5.7 Work with finer-grained parallelism

More recently at PARC, Frank Crow has experimented with using groups of workstations to compute single images, rather than animation. The distribution is done with the Xerox Compute server (described above). Instead of decomposing the problem by dividing the image up (as most approaches presented above), Crow rendered individual objects in the scene on different processors. These objects must be linearly separable (see [2]), so the method is restricted to '2.5D graphics'. The motivation for this method is that it is easier to predict the time to render a given object rather than the time to compute a slice of an image.

The system was initially tested on images with a small number of linearly separable shapes. These were sorted by depth on the "home" (dispatcher) machine and sent to other worker machines for rendering. Each worker would render the pixels in the bounding rectangle of the shape, and return this image along with a coverage mask for the shape[8]. Finally, the images were composited together on the dispatcher machine.

The improvements gained by distributing the work this way were not substantial. Some statistics of the system's operation were gathered, such as which processor got what job, how long it took, and how much time was spent compositing the images together. This revealed three important things: 1) Some shapes took much longer to render than others; 2) The processes were unevenly distributed to the processors; and 3) The final compositing phase was taking long enough to prevent dramatic speedups on complex images. The process distribution itself was also a source of overhead, as data files had to be shipped out and images collected.

Some steps were taken to improve the benefits of distributing the work. Since the disparity between rendering times for different objects was much larger than expected, some heuristics were developed for estimating the cost of rendering an object, and allowing it to be rendered in several strips. The compositor was also substantially optimized, reducing a major bottleneck. With these improvements in place, Crow was able to improve the parallelism to over 30%. Crow describes the system as "Work in progress" — it has no doubt improved substantially since the paper was written.

6 The Distrib System

At the University of Utah Computer Science Department a system called *Distrib* (developed by Rod Bogart, Glenn McMinn and the author) is currently in use for distributing animation computations over a large network of workstations. The computing environment used by Distrib consists of a large network of workstations, including Apollos, Suns, and a large number of Hewlett Packard Series 9000/300 machines. All of these machines are accessible over the Ethernet using the TCP/IP protocols, and all run some variant of Unix. A Vax 11/785 with a large amount of disk space serves as the central dispatcher machine where Distrib runs.

6.1 Operation

Distrib reads as input two files, one containing a list of jobs to execute and the other a list of machines to execute them on. The *job* file specifies for each job the input and output data files to use, the script to execute on the remote machine, and the parameters for that job (scanlines to render, frame number, rendering options, etc.). The *machines* file describes where the files (programs, texture maps, data files, etc.) live on each machine, and also specifies any restrictions on the use of a given machine. Machines can be set up in three ways:

Unrestricted Distrib uses the machine without reguard to time of day or if somebody is logged in. This mode is used for lightly used machines, where the additional rendering job doesn't cause a major impact. (Users of the machine are always able to kill the job if it does get in the way).

Unoccupied Distrib only uses the machine if nobody is logged in or running a "screensaver" program.

Night only Like unoccupied, except that if Distrib finds the machine in use it won't even check back until the evening (or weekend).

As explained below, it's possible to change these restrictions while Distrib is running.

When Distrib starts it reads in the job and worker machine description files. For each worker, Distrib copies the appropriate data files to the worker, using the Unix *rcp* program. It then uses the *rexec* routine to start the computation on the worker. *Rexec* returns a socket file descriptor that listens to the remote process on the worker machine. Once all of the hosts are started, Distrib listens to all of the *rexec* connections simultaneously with the *select* system call.

When the *select* call returns, (indicating activity on one or more of the sockets) Distrib looks at the messages returned by the worker machines. If the message indicates successful completion of the job, Distrib collects the results from worker (verifying the transfer) and cleans up the data area on the worker. If the worker machine was specified as restricted, Distrib makes sure the machine is still available (i.e., nobody has logged in) before starting another job on it.

If the message from the worker's socket indicates failure (e.g., the process is terminated by somebody logging in, the job stops unexpectedly with an error, or the socket simply closes because a worker crashes) Distrib acts according to the machine's restriction. If the machine is "unrestricted", Distrib marks it as "down", and waits an hour before trying to re-use it. If a machine is marked as "restricted", Distrib marks it as "occupied" and waits until it is free before trying to use it again. In any case, the aborted job is re-queued on the next free machine. Distrib maintains an extensive log of all of this activity.

6.2 Problems encountered

Several interesting problems were encountered in the process of getting Distrib to run reliably. In the original version an *rsh* process was forked to start the remote process instead of using *rexec*. Instead of returning a socket, this returned a process ID and the *wait* system call was used to detect when jobs were finished. While simple to implement, this uncovered a number of problems. Most noticeable was that because each *rsh* created two processes, the Distrib program quickly exceeded the Unix limit of the number of processes a user is allowed to have when a large number of workers were used.

In the original versions of Distrib, a job's input data was copied *from* the dispatcher by each of the individual workers. Distrib would spawn the all workers simultaneously, and they would all start asking the dispatcher for data at the same time. This flooded the dispatcher with I/O requests, and often some of the requests would fail because system limits were exceeded. Distrib now copies the necessary files *to* the workstation before starting the job on it. This serializes the I/O, and prevents the dispatcher from being swamped with file requests.

Because TCP/IP is a "reliable" protocol, connections will not time out once they are initiated. In one case, a worker crashed as the data files were being copied to it, and Distrib became hung waiting for the transfer to complete, preventing it from starting work on other machines. It now sets up a "watchdog" timer before sending or retrieving files from workers. If the transfer doesn't complete before the timer runs out, Distrib receives a signal and the job is aborted. The worker is marked as "down" and the job is re-queued.

6.3 Interaction with Distrib

A Distrib run may last for several days. During this period of time, it's useful to be able to interact with Distrib to inquire about the status of the jobs or to make minor adjustments to its state. To accommodate this, Distrib listens to a "command" socket in addition to the *rexec* sockets. When a connection is made to this socket (usually with a utility like telnet) the user can interact with Distrib and find out exactly what the status of the computation is. This is usually much quicker than trying to get the same information from Distrib's log files, which become quite large during a long run.

Another use for the command socket is to change the state of the machines Distrib is controlling. For example, an unrestricted machine can be changed to restricted if Distrib was interfering with its normal use, or a recently re-booted machine can be changed from down to up.

6.4 What if Distrib dies?

The advantage to using the *rexec* connection is Distrib knows exactly when a workstation finishes (or aborts). There is no periodic polling needed to get a worker's status. A disadvantage to this approach is that most of Distrib's state is in the form of open *rexec* sockets. If the machine Distrib is running on goes down, there is no way to recover this information. When the dispatcher dies, the usual approach is to wait an hour or so for most of the workers to finish their jobs.[4] Scripts are then run to collect any finished work, kill any remaining jobs, and clean

[4] i.e, go out for pizza...

up the worker data directories. A new job file is made by subtracting the finished work from the original job file, and Distrib is re-started.

This problem could be solved by making the system running on the worker end more intelligent. The worker would have two processes, one to do the rendering and the other to talk to Distrib. This second "supervisor" process could detect if Distrib went away, and listen for a new Distrib if it did. When Distrib is restarted, it would contact all of the supervisor processes, determine their state, and pick up the computations based on this information. Fortunately, the Vax Distrib is usually run on has proved quite reliable, so motivation to implement this scheme has been low.

6.5 Some Results

Some example Distrib runs include:

- A high resolution still of a butterfly. The image was computed at 1024x1024 pixel resolution. The jobs consisted of ray-tracing 32 horizontal strips of the image. It took three hours elapsed time using 13 idle HP Series 9000/320 machines. The total CPU time used was 27 hours, so the computation achieved about 70% efficient parallelism.

- A simple animated station logo (approximately two seconds worth). It took 24 hours elapsed time using 30 workstations. The total CPU time used was 24 days, 10 hours (about 80% efficient).

- Another run for producing twelve seconds of animation took 64 hours of elapsed time. It ran on 60 workstations (some of them un-available during the day). The total CPU time was three months, 3 days and 20 hours (2252 hours), about 57% efficient.

In most of these cases, production deadlines were met that would not have been possible without a facility like Distrib.

7 Conclusions

There are some consistent features in the systems presented above. Almost all of them provide facilities for logging the activity performed. Since the computation involved often extends over hours or days, there is no other way to supervise the work. Log files are often the only way to debug the system when it's actually in use.

Fault tolerance is an important issue. Even if the average reliability of a machine is good (say, only one shutdown or failure a month), this decreases rapidly as you use more machines (e.g., 30 machines gives you one failure a day). Without at least some facility for dealing with worker failures, a distributed computation system often grinds to a halt.

The computing resources offered on a typical large workstation network are substantial — often equivalent to a single supercomputer. Since computer animation is an easily decomposable and large-grained problem, it makes an ideal problem for solving with distributed computation.

8 Acknowledgements

I would like to thank the many people who responded to the Usenet survey and took the time to write up their experiences, Jules Bloomenthal at Xerox PARC for providing information about recent work there, and Jay Lepreau for pointing out recent work with Unix. Glenn Mcminn and Robert Mecklenburg gave the paper a good critical reading.

Peter Ford, Mark Bradakis, and "Charlie Root" provided us with valuable assistance while getting Distrib running.

We would also like to thank the Hewlett Packard corporation for their generous gift of HP workstations. These systems allowed us to work on a very large scale.

This work was supported in part by DARPA (DAAK1184K0017) and the National Science Foundation (MCS-8121750). All opinions, findings, conclusions or recommendations expressed in this document are those of the author and do not necessarily reflect the views of the sponsoring agencies.

References

[1] F. C. Crow. *Experiences in Distributed Execution: A Report on Work in Progress.* Tutorial Course Notes: Advanced Image Synthesis, ACM-SIGGRAPH, August 1986.

[2] F. C. Crow. A more flexible image generation environment. *Computer Graphics*, 18(3), July 1984.

[3] T. H. Dineen, P. J. Leach, N. W. Mishkin, J. N. Pato, and G. L. Wyant. The network computing architecture and system. In *Proc. 1987 Summer Usenix Conferance*, Usenix, June 1987.

[4] R. Hagmann. Process sever: sharing processing power in a workstation environment. In *6th Intl. Conf. on Distributed Computing*, IEEE, Cambridge, MA, May 1986.

[5] M. J. Litzkow. Remote unix — turning idle workstations into cycle servers. In *Proc. Summer Usenix Conferance*, Usenix, Phoenix, AZ, June 1987.

[6] Michael Muuss. *Solid Modeling System and Ray-Tracing Benchmark.* Distribution Release Notes, U.S. Army Ballistics Research Lab, December 1986.

[7] John W. Peterson. *A System For High Quality Image Synthesis.* CS Project Memo 86-01, Unversity of Utah, June 1984.

[8] Thomas Porter and Tom Duff. Compositing digital images. *Computer Graphics*, 18(3):253, July 1984. Proceedings of SIGGRAPH 84.

[9] John F. Shoch and Jon A. Hupp. The worm programs - early experience with a distributed computation. *Communications of the ACM*, 25(3):172, March 1982.

[10] Turner Whitted. *The Hacker's Guide to Making Pretty Pictures.* Tutorial course notes – Image Rendering Tricks, ACM-SIGGRAPH, August 1986.

Ray Tracing on the Connection Machine System

Hubert C. Delaney
MIT Media Laboratory
sphere@media-lab.mit.edu

ABSTRACT

Ray-tracing has been proven to be an invaluable tool for realistic image synthesis and the modelling of complex lighting effects. Unfortunately, conventional ray tracing programs typically require large amounts of computation time. It is clear, however, that a certain amount of natural parallelism exists in ray tracing because of the independence of the computations for separate rays. Rays may be assigned to individual processors and traced simultaneously. In SIMD parallel processor implementations, however, objects in the database must be individually broadcast to the processors, resulting in long execution times for large databases. We introduce a parallel implementation of an incremental ray tracing system in which processors consider voxel elements lying along rays. The processors may then simultaneously request information about the contents of their voxels via messages sent through the hypercube network. In this way, N rays may be traced through a database consisting of up to N objects in a short time where N is the number of available processors.

RASTER IMAGE ROTATION AND ANTI-ALIASED LINE DRAWING

Ephraim Cohen

Computer Graphics Laboratory
New York Institute of Technology
Old Westbury, NY 11568

Abstract

This paper presents an algorithm for rotating and scaling a raster image into a raster destination. The algorithm also may be used to draw anti-aliased lines on a raster display.

CR Categories and Subject Descriptors: I3.5 **[Computer Graphics]:** Picture/Image Generation-*display algorithms*

General Terms: Algorithms

Key Words and Phrases: Compositing, matte channel, Line Drawing, Anti-aliasing, Raster displays

1. History

The problem of drawing a slanted line on a raster device goes back to the early use of pen plotters and numerically-controlled machine tools. An early treatment (the DDA, or down-down across, algorithm) is given in [Bresenham 1965], and papers dealing with variations of this algorithm have appeared regularly since.

Two-dimensional transformation of pictures has also been an active area of development, especially for video applications. The AMPEX ADO is a particularly impressive solution to the picture rotation problem--complete video images are transformed in 1/15th of a second at real-time rate. Of course, this uses special-purpose hardware in addition to a clever algorithm--basically that of [Catmull & Smith 1980]. The transformations presented here do not do perspective transformations, as does the ADO. However, the algorithm presented here is self-matting.

2. Objectives

Rotating raster images has always been a cumbersome problem. Rotated pixels do not line up nicely with the destination raster. The simplest method for rotating a raster image is to traverse the destination raster in scanline order, and to calculate the source location and color corresponding to each pixel in the destination. Unfortunately, this means that the source picture is accessed in an essentially random order. Pictures stored as files can not be accessed at random with any efficiency. It is much better to access such pictures in the order in which they are stored--in this treatment, pictures are assumed to be stored in scanline order(left to right, then top to bottom, of the picture). This permits a composite image to be built up in a frame store, without the need for an auxiliary frame store to hold the source picture. The source picture can be accessed directly from its file.

This paper presents an algorithm for rotating and scaling a raster image into a raster destination, with the source image accessed in scanline order. Of course, this means that the destination image pixels must be visited in a more-or-less random order. The method is

similar to that of [Braccini 1980] and [Weiman 1980].

The algorithm may also be used to draw anti-aliased lines. A line may be considered to be a very narrow rectangular picture that is all the same color. The same scheme that rotates a raster image rotates this narrow picture into an anti-aliased line.

3. Assumptions and Notation

We will use u,v coordinates for the source picture, and x,y coordinates for the destination picture. No particular note will be made of whether numbers are fixed- or floating-point, but in fact the implementation has been done using scaled integers.

The transformation we will use is the general affine transformation

$$u = u_x x + u_y y + u_0$$
$$v = v_x x + v_y y + v_0$$

This actually is the inverse of the transformation we want to use. It gives points in the source $[u,v]$ in terms of points in the destination $[x,y]$. This transformation is assumed to be non-degenerate, that is, $u_x v_y - v_x u_y$ is not zero.

We also assume that the source image is rectangular, and contained in the region $0 < u \le u_{max}$ and $0 < v \le v_{max}$. Source pixels inside the region may be partly or wholly transparent, but we must be sure that pixels outside the above rectangle are completely transparent.

The notation for the color of a pixel will be introduced when it is needed.

We also make two definitions. They will be useful for discussing line drawing onto a raster display.

Definition 1. The point $[x,y]$ is *near* a line if the line intersects at least one of the four line segments going from $[x,y]$ to the four points $[x+1,y]$, $[x-1,y]$, $[x,y+1]$, and $[x,y-1]$.

Definition 2. The sequence of points $[x_i\ y_i]$ *sketch* the line

$$U = u_x x + u_y y + u_0 \qquad u(x, y) = u_x x + u_y y + u_0 = U \ (U \text{ constant})$$

if

1. x_i and y_i are integers for all i.

2. There is an integer k such that one of the following four statements is true for all values of i:

$$x_i = k + i \quad x_i = k - i \quad y_i = k + i \text{ or } y_i = k - i$$

(This keeps the points $[x_i, y_i]$ close together).

3. $[x_i, y_i]$ is *near* the line $U = u_x x + u_y y + u_0$.

The word *sketch* describes the output of Bresenham's algorithm as it is used in this paper. See Figure 1.

4. Jagged Line Drawing Algorithm

First, we need an algorithm for sketching the line $u(x, y) = U$ with U constant, in the direction of increasing v, given we have a starting point $[x,y]$ near the line.

If the sequence $[x_i, y_i]$ sketches the line

$$u_x x + u_y y + u_0 = U$$

and we also have

$$v(x, y) = v_x x + v_y y + v_0$$

then the sequence sketches the line in the direction of increasing v provided that

$$v_x x_i + v_y y_i \ge v_x x_{i-1} + v_y y_{i-1}$$

for all values of i.

We now give the line drawing algorithm:

L.1. [Initialization]

[The following four conditional statements refer to the quadrant of the plane the vector $[u_x, u_y]$ is in. The subscripts s and g mean we use the values as increments when the line function $u(x, y)$ is *smaller* or *greater* than its desired value U].

If $u_x < 0$ and $u_y \geq 0$, then set

$$u_g = u_x \quad v_g = v_x \quad x_g = 1 \quad y_g = 0 \text{ [Increment x]}$$
$$u_s = u_y \quad v_s = v_y \quad x_s = 0 \quad y_s = 1 \text{ [Increment y]}$$

If $u_x \geq 0$ and $u_y \geq 0$, then set

$$u_g = -u_y \quad v_g = -v_y \quad y_g = -1 \quad y_g = 0 \text{ [Decrement y]}$$
$$u_s = u_x \quad v_s = v_x \quad x_s = 0 \quad x_s = 1 \text{ [Increment x]}$$

If $u_x \geq 0$ and $u_y < 0$, then set

$$u_g = -u_x \quad v_g = -v_x \quad x_g = -1 \quad y_g = 0 \text{ [Decrement x]}$$
$$u_s = -u_y \quad v_s = -v_y \quad x_s = 0 \quad y_s = -1 \text{ [Decrement y]}$$

If $u_x < 0$ and $u_y < 0$, then set

$$u_g = u_y \quad v_g = v_y \quad y_g = 1 \quad y_g = 0 \text{ [Increment y]}$$
$$u_s = -u_x \quad v_s = -v_x \quad x_s = 0 \quad x_s = -1 \text{ [Decrement x]}$$

L.2. [Modify initialization to force v increasing]

If $v_s |u_g| + v_g |u_s| < 0$, then swap u_g with u_s, and do the same for v, x, and y.

L.3. [Loop initialization]

We are given $[x_0, y_0]$ as a starting point for drawing the line. Compute

$$u = u(x_0, y_0) = u_x x_0 + u_y y_0 + u_0$$
$$v = v(x_0, y_0) = v_x x_0 + v_y y_0 + v_0$$

and set $i = 0$.

L.4. [Sketch generating loop]

If $u < U$, then set

$$u = u + u_s \quad v = v + v_s \quad x_{i+1} = x_i + x_s \quad y_{i+1} = y_i + y_s$$

L.5.

If $u \geq U$, then set $u = u + u_g \quad v = v + v_g \quad x_{i+1} = x_i + x_g \quad y_{i+1} = y_i + y_g$

L.6. [Loop Test]

Set $i = i + 1$

If more points of the sequence $[x_i, y_i]$ are desired, continue at step L.4.

This completes the line drawing algorithm.

This form of the algorithm does more computation than is absolutely necessary. For any particular line, we know that:

1. One of x_s, y_s is zero.
2. One of x_g, y_g is zero.
3. One of the conditional statements in L.4 and L.5 is always true.

The algorithm is usually written to avoid doing the unnecessary operations. However, for the applications discussed here, the speed gained does not seem to warrant the extra complication.

We now find a bound on the range of u-values encountered on the sketched line. Because the sketched points approximate the line $U = u_x x + u_y y + u_0$, the range of u-values at the points $[x_i, y_i]$ is quite restricted. For any sketched point $[x_i, y_i]$ of the line, there is a point $[x, y]$, not necessarily an integer point, exactly satisfying

$$U = u_x x + u_y y + u_0$$

and such that either

$$x = x_i \quad \text{and} \quad |y - y_i| \leq 1$$

or

$$y = y_i \quad \text{and} \quad |x - x_i| \leq 1$$

[This is a restatement of the definition of nearness].

Then the absolute value d of the the difference between U and u at $[x_i, y_i]$ is:

$$\begin{aligned} d &= |u_x x_i + u_y y_i + u_0 - U| \\ &= |u_x x_i + u_y y_i + u_0 - u_x x - u_y y - u_0| \\ &= |u_x(x_i - x) + u_y(y_i - y)| \\ &\leq \textit{maximum of } |u_x| \textit{ and } |u_y|. \end{aligned}$$

Now u_x and u_y are generally small--they are large only if the source picture is being greatly reduced in size. For a pure rotation through angle θ, we have $u_x = \cos\theta$ and $u_y = \sin\theta$, and so the maximum difference is ≤ 1 for any θ.

5. Area Drawing Algorithm

We draw over an area by drawing lines next to one another, so that the sketched images of the lines cover all the integer points of an area of the destination. That is, any integer point $[x, y]$ of the destination that lies under the image of the source picture is on the sketch of some line being drawn. In fact, the algorithm described below puts each integer point of the destination on *exactly* one of the lines drawn. Thus, there are no holes in the image, and no points that are drawn twice. This is accomplished by carefully choosing the spacing of the lines sketched. The proof of this is omitted, but see Figure 2.

The algorithm is as follows:

A.1.

Compute u_g, v_g, x_g, y_g, and u_s, v_s, x_s, y_s as in L.1 - L.2 above. [This initializes the line sketching. It need be done only once because all the lines sketched are parallel].

A.2.

If $|u_x| > |u_y|$, set $du = u_x \quad dv = v_x \quad dx = 1 \quad dy = 0$
Otherwise, set $du = u_y \quad dv = v_y \quad dx = 0 \quad dy = 1$
[This sets the direction we will use for going from one scanline to the next as roughly perpendicular to the direction of the scanline images].

A.3.

If $du < 0$, set $du = -du \quad dv = -dv \quad dx = -dx \quad dy = -dy$
[This forces the lines drawn to have increasing u-values ($du > 0$)].

A.4.

Get an initial $[x_0, y_0]$ such that:

x_0 and y_0 are integers
$u_0 = u_x x_0 + u_y y_0 + u_0 \leq 0$

$v_0 = v_x x_0 + v_y y_0 + v_0 \leq 0$

$[x_0, y_0]$ is *near* the image of the point $[u=0, v=0]$

Such a point may be found by using the forward transformation of $[u, v]$ into $[x, y]$. We give no notation for this transformation because it is only used at this step. [The line drawing algorithm works using differences. This step gives it a place to start].

A.5.

Set $U = 0$ [Initialize the scanline image counter]

A.6. [Start of the outer area-drawing loop]

If $v_0 \leq 0$ continue at step A.7.

A.6.1.

If $u_0 > U$, set $u_0 = u_0 - u_s \quad v_0 = v_0 - v_s \quad x_0 = x_0 - x_s \quad y_0 = y_0 - y_s$

A.6.2.

If $u_0 \leq U$, set $u_0 = u_0 - u_g \quad v_0 = v_0 - v_g \quad x_0 = x_0 - x_g \quad y_0 = y_0 - y_g$

A.6.3

Continue at A.6 above. [This is the line drawing algorithm of the previous section, applied to move backwards along the line until $[u_0, v_0]$ is off the opaque part of the source image, with $v_0 \leq 0$. This is usually the case anyway. Note that destination pixels are not modified at this step].

A.7. [Initialize the line drawing loop]

Set $u = u_0 \quad v = v_0 \quad x = x_0 \quad y = y_0$

A.8. [Main scanline drawing loop]

A.8.1.

If $u < U$, set $u_0 = u_0 + u_s \quad v_0 = v_0 + v_s \quad x_0 = x_0 + x_s \quad y_0 = y_0 + y_s$

A.8.2.

If $u \geq U$, set $u_0 = u_0 + u_g \quad v_0 = v_0 + v_g \quad x_0 = x_0 + x_g \quad y_0 = y_0 + y_g$

A.8.3.

Draw the pixel at $[u, v]$ into the destination pixel at $[x, y]$ (see the next section).

A.8.4.

If $v < v_{max}$, continue at A.8. above.

A.9. [Move to the next scanline to sketch]

Set $U = U + du \quad u_0 = u_0 + du \quad v_0 = v_0 + dv \quad x_0 = x_0 + dx \quad y_0 = y_0 + dy$

A.10. [End the outer loop]

If $U < u_{max}$, go to A.6. above.

This completes the area drawing algorithm.

6. Drawing Each Pixel

This section describes in more detail the step A.8.3 above. We will assume the source image has four coordinates per pixel, values of red, green, blue, and opacity at each integer point of the source, and the destination pixels have three coordinates, red, green, and blue values. We denote these by the seven functions:

$$r(u,v),\ g(u,v),\ b(u,v),\ a(u,v)$$

$$R(x,y),\ G(x,y),\ B(x,y)$$

defined for integer values of u, v, x, and y. We also define r, g, and b as having already been multiplied by $a(u,v)$, because this greatly simplifies computations (see [Duff 1984]). Note also that $a(u,v) = 0$ for $u, v < 0$ and for $u \geq u_{max}$ or $v \geq v_{max}$.

At each pixel to draw, the x, y coordinates are guaranteed to be integers, but the u, v coordinates are generally not integers. We must therefore extend the functions r, g, b, a from the integer to the real domain--they must be defined for any real values of u and v. This is difficult to do. If the source image is being made smaller, we want the extended function to average many source pixels together, to avoid aliasing effects in the shrunken destination image. The amount of such blurring of the source that is desirable depends on the values u_x, u_y, v_x, and v_y. Problems of this nature will be disregarded here (see, for example, [Kajiya 1981]). We describe two crude schemes that work for pictures whose size is not being reduced radically.

The simplest scheme, and one that works well if the source image is already somewhat blurry (as are images scanned in from a video source), is to use

$$r(u,v) = r(U_{int}, v_{int})$$

where U_{int} is the integer nearest U (U being the outer loop parameter of the area-drawing algorithm). We know that u is near U by the discussion of the line drawing algorithm. This allows the algorithm to buffer only one scanline at a time. The new destination pixel is

$$R(x,y) = r(u,v) + (1 - a(u,v))R(x,y)$$

(because $r(u,v)$ is already multiplied by $a(u,v)$). $G(x,y)$ and $B(x,y)$ are computed similarly.

Somewhat better results may be had by using bilinear interpolation in the source image. We represent u and v as $u_{int} + u_{frac}$ and $v_{int} + v_{frac}$ (integer and positive fractional parts), and extend the functions from integers to real numbers by taking

$$\begin{aligned} a(u,v) = \ & a(u_{int}, v_{int})(1-u_{frac})(1-v_{frac}) \\ & + a(u_{int}+1, v_{int})u_{frac}(1-v_{frac}) \\ & + a(u_{int}, v_{int}+1)(1-u_{frac})v_{frac} \\ & + a(u_{int}+1, v_{int}+1)u_{frac}v_{frac} \end{aligned}$$

and doing the same for $r(u,v)$, $g(u,v)$, $b(u,v)$ (we can do this only because r, g, b have already been multiplied by $a(u,v)$). Then the new destination pixel is given as above by

$$R(x,y) = r(u,v) + (1 - a(u,v))R(x,y)$$

$G(x,y)$ and $B(x,y)$ similarly.

As we traverse each scanline $u = U$, the u-values wander through the range of values computed in section 3. above. Thus, when using bilinear interpolation, we must keep $2(max(|u_x|, |u_y|) + 1$ scanlines available at the same time. As also shown above, this works out to 3 scanlines for pure rotations. Thus, only a small amount of buffer space is needed for the source image.

The above formula for $R(x,y)$ assumes the six functions r, g, b, R, G, B are linear in luminosity. If they are not, the color functions must be converted to linear luminosities for use in the above formula, and the result of the formula converted back to non-linear values. For video monitors, the values of these functions are generally linear with gun voltages, and the luminosity is given by r^{γ}, where γ is about 2. In this case, the corrected formula is

$$R(x,y) = (r(u,v)^{\gamma} + (1-a(u,v))R(x,y)^{\gamma})^{1/\gamma}$$

This may be done by using prestored tables of x^{γ} and $x^{1/\gamma}$. This correction does not greatly affect the appearance of rotated pictures. However, it is vital for the good

appearance of anti-aliased lines.

Finally, the computation of R, G, B would be incorrect if pixels were modified more than once each, because the source color would be put in with too great an opacity.

7. Application to Anti-aliased Line Drawing

Algorithm A may be used as it stands to draw an anti-aliased line. We simply apply it to a thin rectangular picture that is all the same color. Averaging between the opaque and transparent pixels at the edge of the rectangle will produce anti-aliasing in its rotated image. But better anti-aliasing, and fancier line types may be drawn if the algorithm is modified to take advantage of the simplifications possible when drawing lines. Clearly, line color and shape should be computed, and not prestored as a picture (although for some applications we may store the cross section of a line in an array).

Assume we wish to draw an anti-aliased line from $[x_0, y_0]$ to $[x_1, y_1]$. Before we can use the algorithm, we must compute its arguments. The equations for u and v we will use are

$$u = (y_1-y_0)x - (x_1-x_0)y - x_0y_1 + x_1y_0$$

$$v = (x_1-x_0)y + (y_1-y_0)x - (x_1-x_0)x_0 - (y_1-y_0)y_0$$

from which

$$u_x = y_1-y_0 \quad u_y = x_0-x_1 \quad u_0 = x_1y_0-x_0y_1$$
$$v_x = x_1-x_0 \quad v_y = y_1-y_0 \quad v_0 = -v_xx_0-v_yy_0$$

From this definition, the lines v = *constant* are parallel to the line we wish to draw, and u goes from 0 at $[x_0, y_0]$ to

$$u_{max} = u_x^2 + u_y^2$$

at $[x_1, y_1]$. Because $u_{max} \geq 0$, we may omit step A.3.

We are given $[x_0, y_0]$, and so may also omit step A.4, provided we change the limits on v from $0 \leq v < v_{max}$ to $-wu_{max} < v < wu_{max}$, where w is the width (in pixels) of the line being drawn.

The line width w may be interpolated along the length of the line. At step A.6, we may take

$$wu_{max} = w_0u_{max} + \frac{U}{u_{max}}(w_1-w_0)u_{max} = w_0u_{max} + U(w_1-w_0)$$

where w_0 and w_1 are the line widths at the endpoints of the segment being drawn. Line color may also be interpolated along the line segment at this point in the algorithm, as may any other characteristic of the line we may wish to vary. Note that this saves considerable interpolation for wide lines--one interpolation computes the line parameters for all pixels in each v-loop.

The computation of opacity is critical to the smooth appearance of the displayed line. We use the following scheme for opaque lines:

If $|v| < (w-1)u_{max}$, then the pixel is opaque.
If $|v| \geq wu_{max}$, then the pixel is completely transparent.
Otherwise, the opacity is $A\left(\frac{|v|}{u_{max}} - \left(w-\frac{1}{2}\right)\right)$

where $A(r)$ is a function that gives the area of a circle partly covered by a half-plane at distance r from the center of the circle (see Figure XX). This is the amount of the pixel covered by the line if we assume the pixel is circular--an unjustified assumption, but one that allows us to disregard the orientation of the line at this step. We have

$$A(r) = \frac{4}{\pi}\int_{-1/2}^{r}\sqrt{1-4r^2}\,dr = \frac{1}{2} + \frac{1}{\pi}\left(2r\sqrt{1-4r^2} + \sin^{-1}2r\right) \text{ for } -\frac{1}{2} \leq r \leq \frac{1}{2}$$

Of course, the values of $A(r)$ are precomputed and saved in a table. This is a variation of

the technique described in [Gupta 1981].

Thin lines must be handled somewhat differently. If the line is less than one pixel wide, it seems best to draw the line one pixel wide, and use the width as a maximum opacity--that is, reduce the opacity of the pixels drawn by multiplying opacity by line width. Thus, thin lines are anti-aliased, and the illusion of thinness is given by transparency. This method also gives a smooth transition along a line whose width goes from greater than one pixel to less than one pixel along its length--a necessary condition for acceptability.

Rectangular ends on lines are often undesirable. When we draw a smooth curve by approximating it with straight line segments, rectangular ends on the segments leave little pie-shaped gaps in the curve. We get rid of these gaps, and round off the ends of line segments, by drawing circular spots at all the line segment endpoints. This is a standard technique, but the u-values we carry along for each line allow us to draw pixels only in the pie-shaped areas left uncovered by the lines. In particular, we draw a pixel of the circular spot where two line segments join if at that pixel:
1. The u-value of the previous line equation is greater than u_{max} of the previous line, and
2. The u-value of the next line equation is less than 0.
Thus, keeping track of two u-values as we traverse the circular spot permits us to avoid recomputing pixels.

8. Conclusion

This algorithm provides efficient implementations of its two objectives--two-dimensional picture transformation and drawing anti-aliased lines. It is efficient because it rapidly finds all those pixels that must be redrawn, and finds each such pixel exactly once. This is important because the recomputation of pixels is the most time-consuming part of any such algorithm, as may be seen by consideration of the baroque bilinear interpolation formula of section 6. When it is recalled that this formula must be applied four times for each pixel (to compute r, g, b, and a), it is easily seen to require many more operations than the remainder of the algorithm.

Versions of these algorithms have been implemented in C under UNIX on VAX and PDP11 computers, and also in ADAGE 3000 microcode.

9. Acknowledgments

Thanks to Tom Shermer for fixing my bug-ridden version of this program, and for moving it to ADAGE 3000 microcode. The program is very sensitive to incorrect conditional tests, and is something of a trial to the implementer.

Thanks also to J. J. Larrea and Patrick Hanrahan for their suggestions and corrections of the drafts of this paper.

10. References

Braccini, C. and Marino, G Fast Geometrical Manipulations of Digital Images *Computer Graphics and Image Processing* 13:127-141 1980

Bresenham, J.E. Algorithm for computer control of a digital plotter. *IBM Systems Journal* 4(1):25-30, July 1965.

Catmull, E. and Smith, A. R. 3-D Transformations of Images in Scanline Order *Computer Graphics* 14(3):279-285, July 1980.

Duff, T. and Porter, T. Composing Digital Images i. Computer Graphics 18(3):253-260, July 1984

Gupta, S. and Sproull, R. F. Filtering Edges for Grey-Scale Displays *Computer Graphics* 15(3):1-7, August 1981

Kajiya, J. and Ullner, M. Filtering High Quality Text of Display on Raster Scan Devices *Computer Graphics* 15(3):7-15, August 1981

Pitteway, M.L.V. and Watkinson, D.J. Bresenham's Algorithm with Grey Scale *CACM* 23(11):625-626, November 1980

Weiman, C. F. R. Continuous Anti-Aliased Rotation and Zoom of Raster Images *Computer Graphics* 14(3):286-293, July 1980

Whitted, T. Anti-Aliased Line Drawing Using Brush Extrusion *Computer Graphics* 17(3):151-156, July 1983

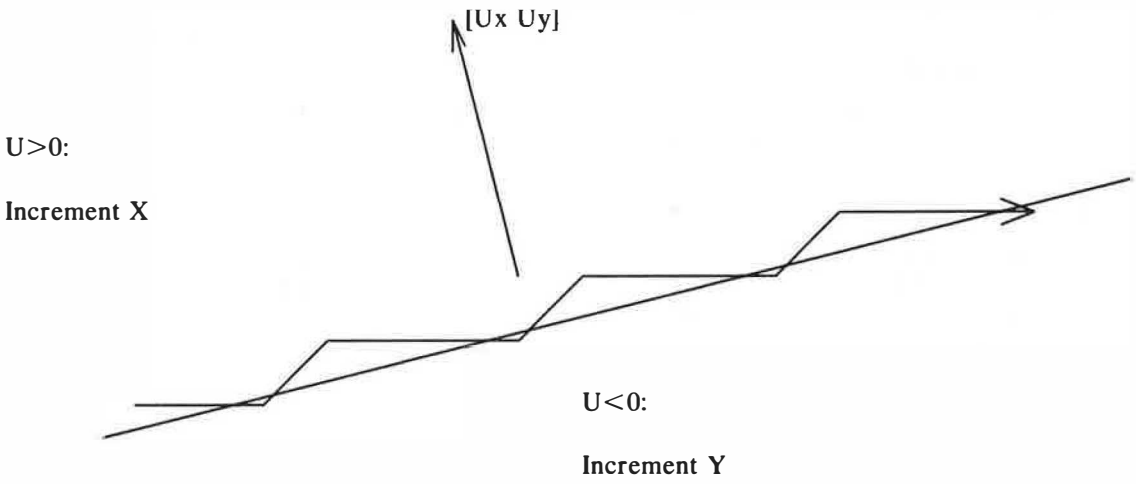


Figure 1. Jagged line drawing.

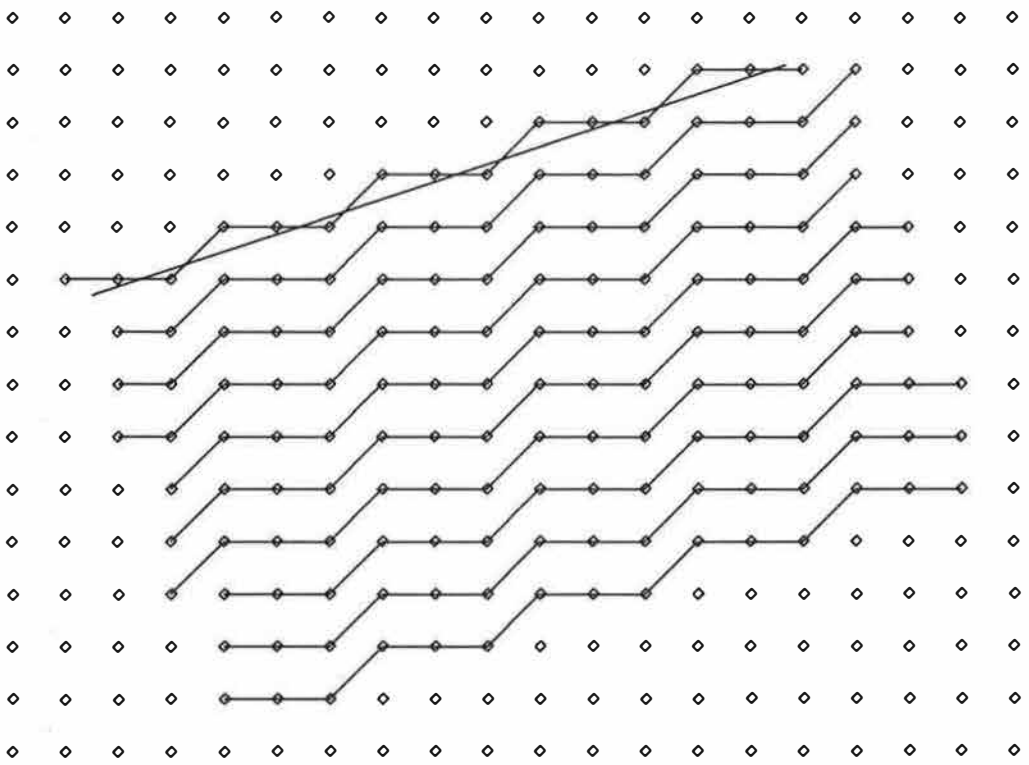


Figure 2. A small rotated rectangle:
The sketched u-constant lines
cover the integer points of the plane
exactly once.

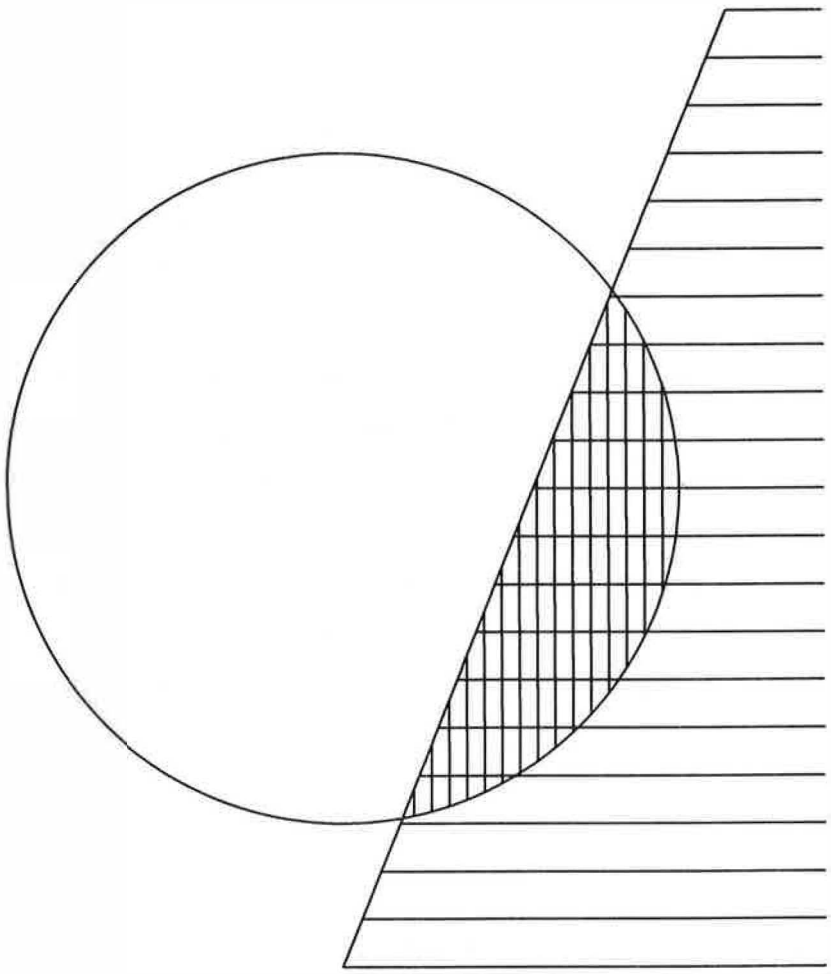


Figure 3. The edge of a line
covers part of a circular pixel.

Dynamics for Everyone

Jane Wilhelms

UCSC-CRL-87-6
May, 1987

Department of Computer Science, University of California, Santa Cruz, CA 95064 USA

ABSTRACT:

There is a move in computer graphics toward more correctly simulating the world being modeled in hopes of achieving more realistic and interesting still images and animation. An important component of this move is use of dynamics, i.e. considering the world as masses acting under the influence of forces and torques. Dynamics an be useful in providing inverse kinematics, constraints, collisions, and, in general, help produce realistic positions and rates of motion. However, it is computationally expensive, involved to program, and complex to control.

1. What is Dynamics and What can it Buy Us?

Dynamics refers to the description of motion as the relationship between forces and torques acting on masses. If we treat the objects modeled in computer graphics as masses and apply forces and torques to them, we can use physics to find out the motion these masses should undergo. This motion should mimic the motion that would actually occur to such masses in the real world, hence dynamics *simulates* the motion, rather than just *animating* it.

Dynamics is useful for a number of reasons: it can help restrict motion to that which is realistic in the world modeled; it can automatically find many kinds of complex motion with minimal user input (e.g., motion due to gravity); it can automatically impose many kinds of constraints (e.g., preventing intersection of colliding bodies); it can be used to move complex bodies in natural way; etc.

Dynamics is problematic as a technique for motion control in computer animation because it is (often) computationally expensive, and because controlling the motion is (often) difficult. However, it shows considerable potential for manipulating and animating bodies, and merits further investigation.

This paper attempts to provide enough basic information to let anyone simulate simple objects using dynamics. A caveat: I'm not a physicist and I haven't had everything here carefully checked by one. It is a culling of relevant information from lots of different sources, which are listed in the references at the back. I would be glad to hear about errors and suggested improvements.

2. How To Do It?

To use dynamics to find the motion of objects, first we must set up the *dynamics equations of motion* which describe how *masses* will move under the influence of *forces* and *torques*. Though there are a number of ways to formulate the equations, they all should give the same solution (they refer to the same world). Second, we must *solve* the equations for *acceleration*. Third, we must *integrate* to find the new *velocity* and *position*, given that acceleration. Once we have the new position, we can animate the object.

There are many books discussing dynamics; unless some specific reference needs to be made, most of the physics in this paper relies upon these references.[7, 10, 17, 19, 21] Robotics books are often useful.[14, 18] The following references pertaining to use of dynamics for computer animation may also be useful.[1, 2, 3, 22, 23, 24, 25, 26]

We will be assuming a right-handed coordinate system with a right-hand screw rule for rotations, and I am assuming that vectors are premultiplied by matrices to change coordinate frames. (This is more in keeping with robotics and physics usage than computer graphics.) Note that considerable variation in conventions are found in the literature; keep in mind which frame and which screw rule you are using. [14, 18]

Matrices will be in uppercase boldface type (**J**), vectors in lowercase boldface (**f**), and scalars in italic type (m). Subscripts will be used to describe the axis for vectors ($\mathbf{c}_x$

This work was supported by National Science Foundation grant number CCR-8606519 and UCSC fellowship 660177-19900.

is the position of the center of mass along the x-axis), and to further describe the value when necessary ($\mathbf{f}_{grv,i,x}$ is the force of gravity acting on the i-th segment along the x-axis). Superscripts will be used to indicate the frame of reference being used, when necessary ($\mathbf{c}^{j}_{x}$ is the above seen in terms of the instantaneous position of the j-th coordinate frame).

Table 1. is a handy reference for the meaning of terms.

2.1. Particles: Point Masses

To illustrate the method on a very simple object, consider the motion of a point mass (a *particle*) in three-dimensions. Dynamics can be done in two dimensions and it's much easier, but also much less interesting.

2.1.1. Information Needed

2.1.1.1. Invariant Information

The only extra piece of constant information we need to dynamically animate particles is the *mass* of the particle. (We could also do dynamics on a particle of changing mass but it's probable that, for computer graphical purposes, constant mass is a reasonable assumption.)

2.1.1.2. Variable Information

Variable data we need for dynamically animating particles includes its present position **p** (a 3d vector representing x,y, and z-coordinates) and its present velocity **v** (also a 3d vector representing the present motion of the particle). (Again, other coordinate systems could be used, but the cartesian x,y,z system seems reasonable.) The fact that we need 3 numbers to specify the position implies that the particle has three degrees of freedom of motion.

We also need to know the force **f** (a 3d vector with components pulling along the x,y, and z-axis) being applied. If a number of forces are pulling at once, we need only add the vectors representing the individual forces to get a net force.

2.1.2. Equations

According to Newton's Second Law, the dynamics of a particle can be stated as

$$\mathbf{f} = m\,\mathbf{a} \qquad 1.$$

where **f** is the force (a 3d vector representing the components of the force along each cartesian axis) acting on the particle, m is the mass of the particle, and **a** is the acceleration that the particle will undergo. Typically, force is in *Newtons* (*kilograms*$-$*meters*$/$*second*^{2}), mass is in *kilograms*, and acceleration is in *meters*$/$*second*^{2}.

This vector equation really represents three scalar equations, one for each cartesian axis. These three equations are

$$f_x = m\ a_x \quad \text{1.a}$$

$$f_y = m\ a_y \quad \text{1.b}$$

$$f_z = m\ a_z \quad \text{1.c}$$

The Second Law Equation is a differential equation, because the acceleration is a function of time. The equation can be also stated

$$\mathbf{f} = \mathrm{m}\frac{\mathrm{d}\,\mathbf{v}}{\mathrm{dt}} \quad \text{2.}$$

because the acceleration is really the derivative (rate of change) of the velocity over time. (The force may also vary with time.) Similarly, it could be stated

$$\mathbf{f} = \mathrm{m}\frac{\mathrm{d}^2\mathbf{p}}{\mathrm{dt}^2} \quad \text{3.}$$

because the velocity is the derivative of the position over time, and, thus, acceleration is the second derivative of the position.

2.1.3. Solving the Equations of Motion

If the user provides the particle mass and the applied force, it is easy to see that solving these three independent equations will give the acceleration that the particle will undergo along each cartesian axis, by dividing by the mass. For example, for x

$$a_x = \frac{f_x}{m} \quad \text{4.}$$

2.1.4. Integrating to Find the New Velocity and Position

The above equations will give us the acceleration, but not the position. A simple method of integrating this equation is referred to as the *Euler method*. It is a numerical (= approximate) solution whose inaccuracy increases as does the acceleration or the time steps used. The Euler method assumes we know the present velocity (e.g. at time i) and want to find the velocity a bit (δt) further on in time; the new velocity will be

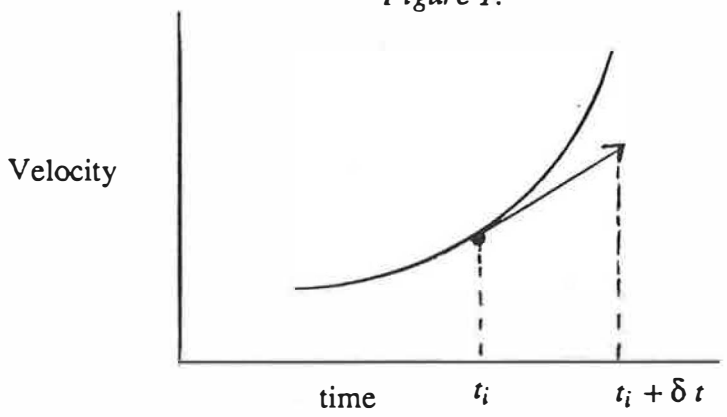
$$\mathbf{v}_{i+1} = \mathbf{v}_i + \mathbf{a}_i \delta t \quad \text{5.}$$

Again, this is really three separate equations. For example, for x

$$v_{i+1,x} = v_{i,x} + a_{i,x} \delta t \quad \text{5.a}$$

This gives us an approximation of the new velocity, but only an approximation. See Figure 1., which represents how the velocity is really changing over time. A point on the curve at time t_i represent the velocity at a particular time t_i. The arrow leaving the curve at a tangent represents the instantaneous acceleration at that time, found from Equation 4. in the previous section. The Euler approximation amounts to moving δt units along the time axis and assumes the new velocity is where the arrow is at time $t_i + \delta t$. Note this is not on the curve. How far off the curve it is depends on how much the curve is bending away from the arrow and how large δt is. With reasonably small time steps we can use this method without too much trouble arising.

Figure 1.



Given the new velocity, we can now find the new position by the same method

$$\mathbf{p}_{i+1} = \mathbf{p}_i + \mathbf{v}_i \delta t + \frac{1}{2}\mathbf{a}_i \delta t^2 \qquad 6.$$

Again, this is really three separate equations. For x,

$$p_{i+1,x} = p_{i,x} + v_{i,x} \delta t + \frac{1}{2} a_{i,x} \delta t^2 \qquad 6.a$$

The same inaccuracy problem occurs when finding the new position. There are better methods of numerical integration, such as the Runge-Kutta method.[5]

2.1.5. Controlling the Motion

Controlling particles is pretty simple. The user need only supply an external force as one 3d vector, or as a normalized (length 1) 3d vector representing the direction of the force and a scalar magnitude representing the strength of the force. It might be desirable to have gravity act on the particle. The gravitational force $\mathbf{f}_{grv}$ is the product of a gravitational acceleration (about 9.81 *meters/second*^{2} on earth, acting toward the earth's center) times the particle mass.

Others forces that might be of interest involve collisions with other objects, and are discussed briefly later on.

2.2. Rigid Bodies: Extended Masses

Assuming that the objects are extended masses, not point masses, complicates things considerably. We assume for now that these extended masses are rigid, and do not change shape or mass.

2.2.1. Information Needed

2.2.1.1. Invariant Information

The constant information that we need includes the mass m of the object, the *center of mass* c of the object (the balance point), and a way to describe how the mass is distributed about the center of mass. The mass is simple.

The center of mass is a 3d vector describing a location in space. This could be a vector from the origin of the world (inertial) space within which all objects are placed, but then we would have to keep changing it as the object moved. It is better to assume some *local* coordinate frame fixed to the object and describe the center of mass relative to this local frame. As long as we know where the local frame is relative to the world frame, it is easy to find the world space center of mass if necessary. Typically such a local frame is already used to describe the geometry of objects for graphics. If the center of mass is not known, picking a point roughly at the center of the object generally is sufficient.

Describing the mass distribution can be more complex, particularly if the object is not symmetrical. Mass distribution for symmetrical objects requires three *moments of inertia*, one about each axis.

$$I_x = \int(y^2 + z^2)dm \qquad 7.a$$

$$I_y = \int(x^2 + z^2)dm \qquad 7.b$$

$$I_z = \int(x^2 + y^2)dm \qquad 7.c$$

i.e., the sum of the masses of each particle making up the object (dm) multiplied by the square of its perpendicular distance from the axis.

For symmetrical bodies there are simple ways of calculating these moments of inertia. For example, for a box centered at the origin with width c in x, b in y, and a in z, the moments of inertia around the origin are

$$I_x = \frac{1}{12}m(a^2 + b^2) \qquad 8.a$$

$$I_y = \frac{1}{12}m(a^2 + c^2) \qquad 8.b$$

$$I_z = \frac{1}{12}m(b^2 + c^2) \qquad 8.c$$

Often this bounding box is a close enough approximation.

If the object is not symmetrical, the three *products of inertia* must also be found. For objects symmetrically arranged around a center of mass, the products of inertia relative to the center of mass are all zero. The products of inertia are shown below. (Note that occasionally products of inertia are predefined as negative quantities, making terms involving them change sign in the dynamics equations.)[10]

$$I_{xy} = \int xy \, dm \qquad 9.a$$

$$I_{xz} = \int xz \, dm \qquad 9.b$$

$$I_{yz} = \int yz \, dm \qquad 9.c$$

The units for moments and products of inertia in the metric system are *kilogram–meters*^{2}.

Often the moments and products of inertia are arranged in a 3x3 *inertial tensor* matrix for using in the equations of motion.

$$\mathbf{J}_k = \begin{bmatrix} I_x & -I_{xy} & -I_{xz} \\ -I_{xy} & I_y & -I_{yz} \\ -I_{xz} & -I_{yz} & I_z \end{bmatrix} \qquad 10.$$

Estimating the moments of inertia for simple symmetrical bodies is simple. It is also quite straightforward to find the moments and products of inertia about any axes or points in space given this information. For example, if you should want these values for the axes of a second coordinate system whose major axes are parallel to the local frame but displaced by $(\delta x, \delta y, \delta z)$, the new values are

$$I'_x = I_x + m(\delta y^2 + \delta z^2) \qquad 11.a$$

$$I'_y = I_y + m(\delta x^2 + \delta z^2) \qquad 11.b$$

$$I'_z = I_z + m(\delta x^2 + \delta y^2) \qquad 11.c$$

$$I'_{xy} = I_{xy} + m\,\delta x\,\delta y \qquad 12.a$$

$$I'_{xz} = I_{xz} + m\,\delta x\,\delta z \qquad 12.b$$

$$I'_{yz} = I_{yz} + m\,\delta y\,\delta z \qquad 12.c$$

Suppose that the new frame isn't parallel to the old. Note that this case may avoidable in your simulations, however, it is worth examining. Equations 11. and 12. take us to a new frame f' whose origin is the same as the desired rotated frame f''. Now we need to find the values for the rotated frame. To do this we need to find the *direction cosines* describing how the new x-axis is related to the old x-axis (a_{00}, a_{10}, a_{20}), the new y-axis to the old y-axis (a_{01}, a_{11}, a_{21}), and the new z-axis to the old z-axis (a_{02}, a_{12}, a_{22}). 21, 15

We can think of the 3x3 rotation matrix representing the orientation of a frame as 3 direction cosine (column) vectors defining the axis of the frame. Column 0 represents the new x-axis, column 1 the new y-axis, and column 2 the new z-axis. To convince yourself of this relationship, try transforming the original axis vectors ((1,0,0),(0,1,0),(0,0,1)) by the rotation matrix.

$$D_k = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} \qquad 13.$$

Now, the new moments and products of inertia (Ix'', etc.) given those found above in a frame parallel to that centered on the center of gravity (Ix', etc.) are

$$I''_x = I'_x a_{00}^2 + I'_y a_{01}^2 + I'_z a_{02}^2 - 2I_{xy} a_{00} a_{01} - 2I_{xz} a_{00} a_{02} - 2I_{yz} a_{01} a_{02} \qquad 14.a$$

$$I''_y = I'_x a_{10}^2 + I'_y a_{11}^2 + I'_z a_{12}^2 - 2I_{xy} a_{10} a_{11} - 2I_{xz} a_{10} a_{12} - 2I_{yz} a_{11} a_{12} \qquad 14.b$$

$$I''_y = I'_x a_{20}^2 + I'_y a_{21}^2 + I'_z a_{22}^2 - 2I_{xy} a_{20} a_{21} - 2I_{xz} a_{20} a_{22} - 2I_{yz}\, a_{21}\, a_{22} \qquad 14.c$$

Similarly, the products of inertia are

$$I''_{xy} = (a_{00}a_{11} + a_{01}a_{10})I'_{xy} + (a_{00}a_{12} + a_{02}a_{10})I'_{xz} + (a_{01}a_{12} + a_{02}a_{11})I'_{yz} \qquad 15.a$$
$$- (a_{00}a_{10}I'_x + a_{01}a_{11}I'_y + a_{02}a_{12}I'_z)$$

$$I''_{xz} = (a_{00}a_{21} + a_{01}a_{20})I'_{xy} + (a_{00}a_{22} + a_{02}a_{20})I'_{xz} + (a_{01}a_{22} + a_{21}a_{02})I'_{yz} \qquad 15.b$$
$$- (a_{00}a_{20}I'_x + a_{01}a_{21}I'_y + a_{02}a_{22}I'_z)$$

$$I''_{yz} = (a_{10}a_{21} + a_{11}a_{20})I'_{xy} + (a_{10}a_{22} + a_{20}a_{12})I'_{xz} + (a_{11}a_{22} + a_{12}a_{21})I'_{yz} \qquad 15.c$$
$$- (a_{10}a_{20}I'_x + a_{11}a_{21}I'_y + a_{12}a_{22}I'_z)$$

This may seem like a drastic amount of trouble, but actually it can be programmed as subroutines and made invisible to the user. In fact, approximate quantities can be found by merely providing a boundary box around the center of mass and assuming some default density to the material (e.g. 1 *kilogram/meter*^{3}). The dimensions of the boundary box (a,b,c) can be used to find the volume ($a \times b \times c$ *meters*^{3}). Multiplying the density by the volume gives the mass. The center of mass can be assumed to be the center of the bounding box. The moments of inertia around the center of mass can be found from Equation 8. above; the products of inertia will be zero. If the frame not at the center of mass but translated away from it, Equations 11. and 12. can be used to find the moments and products of inertia relative to this new frame. If the frame is rotated, Equations 13. and 14. can be used to find the new moments and products of inertia.

2.2.1.2. Variable Information

Rigid bodies have six degrees of freedom. Three are the translational degrees of freedom as with point masses. Three are rotational degrees of freedom describing how the body is oriented toward some frame of reference. Assuming a local coordinate frame fixed to the object, the translational degrees of freedom may represent displacement relative to a fixed inertial world frame axes, or along the present local frame axes (or any other axes). Similarly, the orientation degrees of freedom may refer to rotation about the world space axes, or about the present local frame axes.

We assume the order of rotations will be fixed as x-rotation, then y-rotation, then z-rotation. This means rotations are *Euler*. Euler rotations can come in various orders, here we follow the order x, then y, then z, so that the x-rotation is relative to the original x-axis, the y-rotation is about the y-axis created by the x-rotation, and the z-rotation is about the z-axis created by the former two rotations. Amazingly enough, this can also be thought of as a z-rotation, then a y-rotation, then an x-rotation around the original frame. It is often sensible to assume the local z-axis represents the longitudinal axis of the body, when there is an obvious longitudinal axis.

The other variant information involves the forces $\mathbf{f}$ and torques τ that cause motion to occur. If a number of forces are acting on the body, their total translational effect can be found by merely summing them. The center of mass of the body will move translationally as if it were a particle mass influenced by one net force.

A torque is similar to a force, except that it causes a rotational motion about a particular axis. Torques can be represented as 3d vectors describing their components about an x, y, and z-axis. Torque vectors' net action can be found by summing them.

If all forces are applied at the center of mass, they produce no torque; however, a force acting at a point on the body other than the center of mass will also cause a torque. To find a torque about a coordinate frame's axes due to a force $\mathbf{f}\ (f_x, f_y, f_z)$ applied at point $\mathbf{p}\ (x, y, z)$ (both defined relative to this frame), use this equation.

$$\tau = \mathbf{p} \times \mathbf{f} \qquad 16.$$

or, using components

$$\tau_x = f_z\, y - f_y z \qquad 16.a$$

$$\tau_y = f_x\, z - f_z x \qquad 16.b$$

$$\tau_z = f_y\, x - f_x y \qquad 16.c$$

Often we want motion of the rigid body in terms of its body-fixed frame, and the point of application of the force is in terms of this frame, but the external force is more naturally given in terms of the world inertial frame. An external force (or any other quantity) defined in the inertial frame can be converted into the local frame by multiplying it by the matrix defining how the world frame is oriented as seen from the local frame. This matrix is the inverse (= transpose) of the matrix defining how the local frame is defined relative to the world frame.

If multiple forces and torques are acting upon a body, these six important net values (3 force, 3 torque) can be easily found (for motion relative to the local frame) by summing the forces (in local terms) to find the net **f**, finding the torques caused by these forces using Equation 15., and summing these torques with any active pure torques to find the net torque (τ). This effectively removes the torque component from the forces. After this is done, the net force effectively is applied to the origin of the local frame. The local frame need not be at the center of mass for this to be true.

2.2.2. Equations

With rigid bodies, dynamics becomes somewhat less trivial. There are a number of formulations, and here a brief description of the Euler method is presented. The Euler method is, perhaps, one of the more intuitive formulations. The Armstrong method for articulated body dynamics presented in the next section can, of course, also be used for a single non-articulated body.

The Euler method creates six equations: three are the translational equations of motion relating the linear acceleration and mass to the force, and three are the rotational equations of motion relating the angular acceleration and mass distribution to the torque. Altogether, they specify the behavior of the six degrees of freedom of a free rigid body. Much of this discussion comes from Wells.[21]

The 3d vector version of the translational equations describing the motion of the center of mass is familiar, e.g.

$$\mathbf{f} = m\,\mathbf{a} \qquad 17.$$

or, as 3 scalar equations,

$$f_x = ma_x \;\; ; \;\; f_y = ma_y \;\; ; \;\; f_z = ma_z \qquad 17.a,b,c$$

where **f** is the net force and **a** is the linear acceleration of the center of mass relative to inertial space. This is because the center of mass acts as if the whole body mass were located there and all forces are acting at that point. The effect of these forces on rotation comes out in the rotational equations.

The force and linear acceleration could be expressed relative to any axes, e.g. the instantaneous local axis fixed to the body, by taking the proper components. However, they must both be expressed relative to the same frame. This is an important point, if the user inputs the force $\mathbf{f}^w$ relative to the inertial world coordinate frame and wants the linear acceleration $\mathbf{a}^l$ in terms of the local frame, direction cosines (= rotation matrices) can be used to find the components of the worldspace force relative to the local frame. Another way of looking at this is to take the dot product of the force vector (f_x,f_y,f_z) with each axis vector (e.g., for the x-axis, (a_{00},a_{10},a_{20})). The force component along the local x-axis would be

$$f_x^l = f_x^w a_{00} + f_y^w a_{10} + f_z^w a_{20} \qquad 18.$$

The rotational equations for motion about the center of mass are also quite simple, assuming the products of inertia are zero and that *either* the local frame is at the center of mass or the origin of the local frame is fixed in world space . In this case,

$$\tau_x = I_x \dot{\omega}_x + (I_z - I_y)\omega_y \omega_z \quad 19.a$$

$$\tau_y = I_y \dot{\omega}_y + (I_x - I_z)\omega_x \omega_z \quad 19.b$$

$$\tau_z = I_z \dot{\omega}_z + (I_y - I_x)\omega_x \omega_y \quad 19.c$$

where all values are assumed relative to the local body-fixed frame. ω is the angular velocity of the local frame relative to the inertial frame but expressed in terms of local frame axes. $\dot{\omega}$ is the angular acceleration. ω is typically in *radians/second* and $\dot{\omega}$ in *radians/second*^{2}. τ is the torque acting on the body.

Should you not be so lucky, the more general form of the equations is below. All values are relative to a single coordinate frame, which may be an inertial frame, but is (for our case) probably the instantaneous position and orientation of a body-fixed local coordinate frame. **c** refers to the location of the center of mass relative to this frame. **a** refers to linear acceleration of the origin of this frame. All other values are in terms of this frame as well.

$$f_x = m(a_x \quad 20.a$$

$$f_y = m(a_y \quad 20.b$$

$$f_z = m(a_z \quad 20.c$$

$$\tau_x = m(a_z c_y - a_y c_z) + I_x \dot{\omega}_x + (I_z - I_y)\omega_y \omega_z + I_{xy}(\omega_x \omega_z - \dot{\omega}_y) - I_{xz}(\omega_x \omega_y + \dot{\omega}_z) \quad 21.a$$

$$\tau_y = m(a_x c_z - a_z c_x) + I_y \dot{\omega}_y + (I_x - I_z)\omega_x \omega_z + I_{yz}(\omega_y \omega_x - \dot{\omega}_z) - I_{xy}(\omega_y \omega_z + \dot{\omega}_x) \quad 21.b$$

$$\tau_z = m(a_y c_x - a_x c_y) + I_z \dot{\omega}_z + (I_y - I_x)\omega_x \omega_y + I_{xz}(\omega_y \omega_z - \dot{\omega}_x) \quad 21.c$$

2.2.3. Solving the Equations

Again, the Euler method of numerical integration is often adequate to solve the equations. Note the equations are simple to solve in the direct direction, given accelerations find the forces and torques; however, we want to find linear and angular accelerations given forces and torques. We assume we know the present position and velocity values. Thus we have (at worst) six equations in six unknowns ($a_x, a_y, a_z, \dot{\omega}_x, \dot{\omega}_y, \dot{\omega}_z$).

2.2.4. Controlling the Motion

Rigid bodies can be controlled by a combination of applied torques and applied forces. Applied torques cause a rotational motion about the axes they refer to (e.g. the body-fixed local frame) and require a 3d vector. Applied forces involve a 3d force vector (as with point masses) and also a 3d location vector describing where the force is being applied. Typically the location vector will be specified in the local frame.

Net force is found by summing force vectors irrespective of point of application. Net torque is found by taking the torque caused by these forces (using Equation 15.) as well as any pure torques and summing these. These six values are used in the six equations of motion.

3. Articulated Bodies

Articulated bodies can be thought of as rigid segments connected together by joints capable of less than 6 degrees of freedom. There are numerous formulations of the dynamics equations for rigid bodies, but again, they all come down to the same thing. Some possible choices are the Euler equations,[21] the Gibbs-Appell formulation,[12, 17, 25] the Armstrong recursive formulation,[1, 3] and the Featherstone recursive formulation.[6] The Euler method doesn't deal terribly nicely with constraints at joints. The Gibbs-Appell equations, described in appalling detail elsewhere,[25] have been used for graphical simulation but in a non-recursive form that is $O(n^4)$ in complexity. This is computationally untenable, but if a recursive formulation could be found it still might be a reasonable method, as it allows considerable flexibility in designing joints. (You can design bodies that aren't a hierarchical tree structure alone.) The Featherstone method is recursive and linear in the number of joints, and is flexible in the types of joints, so it might be worth looking into.

The Armstrong method is recursive and linear in the number of joints and will be described in some detail here. It has the slight disadvantage that it can only accommodate bodies with freedom of movement relative to the world (6 degrees of freedom from the body tree root and the world) and three rotary degrees of freedom at each joint. Also bodies must be representable as tree structures. This is fine for most animalistic figures, and further constraints can be applied on top of the basic dynamics using external forces or other more devious methods. The Armstrong method has been used in graphics modeling and I am using it at present, using a modified version of code originally provided by Bill Armstrong and Mark Green at the University of Alberta.

The Armstrong method can be thought of as an extension of the Euler equations with multiple segments (connected rigid bodies). Again, there are at most six equations for each joint (one for each degree of freedom of motion). The real difference comes in the components of the torques and forces. We must consider not only applied forces and torques on the segment, but forces perculating down onto the segment from the children segments, and reaction forces at the joint between the segment and its parent. The following equations are described in detail in Armstrong and Greens 1985 paper.[3] They are repeated here in slightly different terms to show their equivalence to the Euler formulations above.

3.1. Information

The same information is needed for articulated bodies made of rigid segments as for non-articulated rigid bodies, plus a tree describing how the segments are connected together. Each segment can have at most one parent and zero or more children. For convenience, the local frame should originate at the proximal (nearer to the root) joint of a segment and the longitudinal axis of the segment should be the local z-axis. If this convention is followed, the third Euler rotation at a joint will always cause a longitudinal rotation.

If simulating people and other animals, biology and biomechanics books are useful sources of information on the nature of organic tissue, dimensions, etc. NASA's book on anthropometry is also a handy reference.[16]

3.2. Armstrong Equations

Again we have six equations, shown below as two vector equations identical to the Euler equations given above. Everything is expressed in terms of the instantaneous location and orientation of the frame of the i-th segment.

$$\mathbf{f}_i = m_i \mathbf{a}_i - m_i \mathbf{c}_i \times \dot{\omega}_i + m_i \omega_i \times (\omega_i \times \mathbf{c}_i) \qquad 22.$$

$$\tau_i = \mathbf{J}_i \dot{\omega}_i + m_i \mathbf{c}_i \times \mathbf{a}_i + \omega_i \times \mathbf{J}_i \omega_i \qquad 23.$$

In Equation 22., the first term on the right comes from the linear acceleration of frame i, the second from the angular acceleration of frame i, and the third term from the centrifugal force due to rotation of the frame. In Equation 23., the first term on the right is the rate of change of the angular momentum, the second is due to the acceleration of the frame. and the third is due to the rotation of the frame.

If the body is articulated, we must also consider the influence of neighboring segments; in any case we may want to consider external applied forces separate from gravity (pushes and pulls). We can break the force up further into

$$\mathbf{f}_i = m_i \mathbf{a}_{grv,i} + \mathbf{f}_{ext,i} \qquad 24.$$

All these are expressed in terms of the i-th local frame. $m\mathbf{a}_{grv,i}$ $(= \mathbf{f}_{grv,i})$ is the force due to gravity acting on the mass of segment i. $\mathbf{f}_{ext,i}$ is the net external force acting on frame i. $\mathbf{f}_{son,i}$ is the net force due to each son of segment i acting on segment i through the joint joining them. $\mathbf{f}_{topar,i}$ is the net force that segment i is applying to its parent. This force is applied by the parent back onto the son to keep the two from separating (as described in Newton's Third Law), so it is negative in this equation.

We can also break the torques acting on segment i into components

$$\tau_i = m_i \mathbf{c}_i \times \mathbf{a}_{grv,i} + \tau_{ext,i} + \sum(\tau_{son,i} + \mathbf{l}_{son} \times \mathbf{f}_{son}) - \tau_{topar,i} \qquad 25.$$

The first term on the right, $m_i \mathbf{c}_i \times \mathbf{a}_{grv,i}$ ($= \tau_{grv,i}$), describes the effect of gravity acting on the center of mass of the segment and causing a torque at the proximal joint. $\tau_{ext,i}$ is the net external torque applied to the segment i. $\tau_{son,i}$ is the torque that a son of segment i is applying to segment i at the joint between them. $\mathbf{l}_{son} \times \mathbf{f}_{son}$ is the torque due to the force a son segment is applying onto segment i. $\mathbf{l}_{son}$ is a vector from the origin of segment i to the joint between segment i and its son son in terms of frame i. $\tau_{topar,i}$ is the torque that segment i is applying to its parent segment. Forces acting directly on segment i are assumed to have been analyzed to find their torque component acting on segment i and this added to the applied external torques τ_i.

Finally, one more vector equation is needed that relates the acceleration of the parent and son segments. The right side describes the acceleration of the son's proximal hinge due to the the acceleration, angular acceleration, and centrifugal acceleration of the parent i. All are in terms of the axes of frame i.

$$\mathbf{a}_{son} = \mathbf{a}_i - \mathbf{l}_{son} \times \dot{\omega}_i + \omega_i \times (\omega_i \times \mathbf{l}_{son}) \qquad 26.$$

One thing to keep in mind is that though the motion is being described in terms of the axes of frame i, the motion is relative to inertial space, not the the parent. That is, we are not talking about the velocity relative to the parent, which may also be moving on its own. We are talking about an inertial motion that includes the motion of the segment about its joint to the parent plus any motion that parent may be involved in relative to the world.

3.3. Solving the Equations Recursively

Because we limit the body to a tree structure, effects of other segments on a particular segment is limited to effects of sons and parent on this segment. This makes it possible to solve the equations recursively. First we must recognize the linear relationship between angular and linear acceleration, and between linear acceleration and the reactive force on the parent. $\mathbf{K}$ and $\mathbf{M}$ are recursive coefficient matrices which relate linear acceleration to angular acceleration ($\dot{\omega}$) and to reactive force on the parent ($\mathbf{f}_{topar}$), respectively. $\mathbf{d}$ includes other constituents of the angular acceleration and $\mathbf{f}'$ includes other constituents of the force on the parent. For each segment i,

$$\dot{\omega}_i = \mathbf{K}_i \mathbf{a}_i + \mathbf{d}_i \qquad 27.$$

$$\mathbf{f}_{topar,i} = \mathbf{M}_i \mathbf{a}_i + \mathbf{f}'_i \qquad 28.$$

Note that the reactive force $\mathbf{f}_{topar,i}$ acting on the parent j of segment i is one of the $\mathbf{f}_{son,j}$ forces seen from this parent (see Equation 24.). By some deft maneuvering described in more detail in Armstrong and Green's 1985 paper, the dynamics equations can be restated using this relationship. The four recursive coefficients for each segment can be found in an inward pass from the leaves of the body tree to the root. Then this information can be used to find the accelerations of each segment from the root back to

the leaves. The root segment has no parent, so it has no reactive force on a parent and Equation 28. can be solved for the root's linear acceleration. This can be used in Equation 27. to find the angular acceleration of the root. This process is repeated outward using the relationship in Equation 26. to find the linear acceleration of the son links and using this to find their angular acceleration.

The actual steps are shown below.[3] Note that $\mathbf{R}^{topar}$ signifies a 3x3 rotation matrix that takes vectors in a local frame into its parent frame, and $\mathbf{R}^{frompar}$ signifies a 3x3 rotation matrix that takes vectors in a parent frame into a son frame, and that these two are transposes of each other. $\mathbf{R}^{toworld}$ signifies a 3x3 rotation matrix that takes vectors in a local frame into the world frame, and $\mathbf{R}^{fromworld}$ signifies a 3x3 rotation matrix that takes vectors from the world frame into a local frame, and these two are also transposes of each other.

It is useful to compute the cross-product operation using a *tilde* matrix. The tilde matrix for a vector $\mathbf{a}$ is a 3x3 matrix that when premultiplied to a vector $\mathbf{b}$ gives the same result as the cross-product $\mathbf{a} \times \mathbf{b}$. It looks like this

$$\tilde{\mathbf{a}} = \begin{bmatrix} 0 & -a_z & a_y \\ a_z & 0 & -a_x \\ -a_y & a_x & 0 \end{bmatrix} \qquad 29.$$

Inward Pass. The inward pass computes the 4 recursive coefficients and some other useful quantities that are used often. (I have slightly simplified this step. Readers are invited to find more quantities to efficiently precompute.) This step can be divided into two passes: one (the *slowband*) need only be done occasionally; the other (the *fastband*) needs to be done each time through the dynamics loop. Remember subscripts indicate which segment the value refers to, and superscripts indicate which frame the value is in terms of (unless it's in frame i). The equations are repeated for each segment. Summations are over all sons of segment i.

The slowband calculations for a segment i are these:

$$\mathbf{a}_{c,son} = \omega_i \times (\omega_i \times \mathbf{l}_{son}) \qquad 30.$$

$$\mathbf{Q}_{son} = \mathbf{R}_{son}^{topar} \mathbf{M}_{son} \mathbf{R}_{son}^{frompar} \qquad 31.$$

$$\mathbf{W}_{son} = \tilde{\mathbf{l}}_{son} \mathbf{Q}_{son} \qquad 32.$$

$$\mathbf{T}_i = (\mathbf{J}_i + \sum (\mathbf{W}_{son} \tilde{\mathbf{l}}_{son}))^{-1} \qquad 33.$$

$$\mathbf{K}_i = \mathbf{T}_i (\sum \mathbf{W}_{son} - m_i \tilde{\mathbf{c}}_i) \qquad 34.$$

$$\mathbf{M}_i = (m_i \tilde{\mathbf{c}}_i)\mathbf{K}_i - m_i \mathbf{I} + \sum (\mathbf{Q}_{son}(\mathbf{I} - \tilde{\mathbf{l}}_i \mathbf{K}_i)) \quad 35.$$

Along the way, we can accumulate some torque and force information for each segment, $\tau_{\sigma,part}$ accumulates torques, and f_σ accumulates forces. Note I'm assuming that external torques ($\tau_{ext,i}$) are being defined in terms of the local frame (and include torques due to external forces), but external forces ($_{ext,i}$) are in terms of the world space frame.

$$\tau_{\sigma,part,i} = -\omega_i \times (\mathbf{J}_i \times \omega_i) + \tau^i_{ext,i} + (m_i \mathbf{c}_i) \times \mathbf{R}_i^{fromworld} \mathbf{a}^{world}_{grv,i} \quad 36.$$

$$f_{\sigma,i} = -\omega_i \times (\omega_i \times (m_i \mathbf{c}_i)) + \mathbf{R}_i^{fromworld} (\mathbf{f}^{world}_{ext,i} + m_i \mathbf{a}^{world}_{grv,i}) \quad 37.$$

The following equations are the fastband, and should be done each time through the dynamics loop loop.

$$\tau_{\sigma,i} = \tau_{\sigma,part,i} - \tau_{topar,i} + \sum (\mathbf{R}^{topar}_{son} \tau^{son}_{topar,son}) \quad 38.$$

$$\mathbf{d}_i = \mathbf{T}_i (\tau_{\sigma,i} + \sum (\mathbf{l}_{son} \times (\mathbf{R}^{topar_{son}} \mathbf{f}'^{son}_{son} + \mathbf{Q}_{son} \mathbf{a}_{c,son}))) \quad 39.$$

$$\mathbf{f}'_i = \mathbf{f}_{\sigma,i} + (m_i \mathbf{c}_i) \times \mathbf{d}_i + \sum (\mathbf{R}^{topar}_{son} \mathbf{f}'_{son} + \mathbf{Q}_{son} (\mathbf{a}_{c,son} - \mathbf{l}_{son} \times \mathbf{d}_i)) \quad 40.$$

Outward Pass: This completes the world traversing the tree inward. Now we traverse the tree outward, again the work can be divided into a slow and fastband depending on whether the information should be updated each time. (Typically I don't differentiate the two.) First the important accelerations of the root segment, the only one capable of translating freely.

$$\mathbf{a}_{root} = -(\mathbf{M}_{root})^{-1} \mathbf{f}'_{root} \quad 41.$$

$$\dot{\omega}_{root} = \mathbf{K}_{root} \mathbf{a}_{root} + \mathbf{d}_{root} \quad 42.$$

For the rest of the segments on the way out to the leaves

$$\mathbf{a}_i = \mathbf{R}_i^{frompar} (\mathbf{a}_{c,i_{par}} + \mathbf{a}^{par}_{par} - \mathbf{l}^{par} \times \dot{\omega}^{par}_{par}) \quad 43.$$

$$\dot{\omega}_i = \mathbf{K}_i \mathbf{a}_i + \mathbf{d}_i \quad 44.$$

$$\mathbf{f}_{topar,i} = \mathbf{M}_i \mathbf{a}_i + \mathbf{f'}_i \quad 45.$$

if needed to check the solution.

Integration: Now we can integrate to find the new positions and velocities. This again consists of a step that needs to be done each time period, and a step that can possibly be done less often.

This step is done each time period. δu signifies an angular change vector accumulating orientation changes. Remember that while these values are defined in terms of the local frame orientation, they are inertial, including motion not only at the joint to the parent but all motion of all ancestors back to the world. For each segment,

$$\omega_{new} = \omega_{old} + \delta t\, \dot{\omega} \quad 46.$$

$$\delta u_{new} = \delta u_{old} + \delta t\, \omega \quad 47.$$

For the root segment, we are also interested in its linear motion. The linear motion of the other segments (here relative to the worldspace frame orientation) can be calculated from their angular motion.

$$\mathbf{v}_{new}^{world} = \mathbf{v}_{old}^{world} + \delta t\, \mathbf{R}^{toworld}\, \mathbf{a}_{new} \quad 48.$$

$$\mathbf{p}_{new}^{world} = \mathbf{p}_{old}^{world} + \delta t\, \mathbf{v}_{new} \quad 49.$$

Finally, we can update the rotation matrices at the slowband rate from distal to proximal (leaves to root). (Reset $\delta \mathbf{u}$ to zero after this operation.)

$$\mathbf{R}_{new}^{topar} = \mathbf{R}_{old}^{topar}(\mathbf{I} + \delta \mathbf{u}) \quad 50.$$

This matrix should be orthonormalized to reduce error accumulation.[8]

Finally, each $\mathbf{R}^{topar}$ and its inverse can be calculated

$$\mathbf{R}_{new,son}^{topar} = \mathbf{R}_{new,i}^{fromworld} \mathbf{R}_{new,son}^{toworld} \quad 51.$$

Armstrong and Green[3] suggest that the numerical instability that sometimes accumulates and causes bodies to flail about can be reduced by reducing the time step δt or by increasing artificially the moments of inertia about longitudinal axes. This latter method may produce some anomalous behavior, however, in my experience.

3.3.1. Control Issues

It is not terribly difficult to write subroutines to do the dynamics explained above (or to borrow the code from a friendly spirit who has done it before you). The open questions involve how to use this dynamic ability to get desirable motion and simulate constraints nicely. Some hints to solving these problems are presented in this section, but a great deal of work remains to be done before we can watch simulated animals moving realistically about on our computer screens under total dynamic control.

Clearly, the way to control the motion is to supply forces and torques that cause or restrict motion, either directly or through sophisticated preprocessors. Control could also be supplied in the form of extra constraint equations that limit the degrees of freedom involved. This method will not be discussed here.

3.4. Automatically Obvious : Gravity

The effect of gravity is easily calculated given the gravitational acceleration (about $9.81 m/sec^2$ on the earth's surface). Assuming the y-axis points away from the center of the earth, the force acting on the center of mass of each rigid body is

$$\mathbf{f}_{grv} = (0, -9.81, 0)\, m \qquad 28.$$

The torque due to this force acting in the body fixed coordinate frame is

$$\tau_{grv} = \mathbf{c} \times \mathbf{f}_{grv} \qquad 29.$$

3.5. External Dynamic Control

The user can shove the body about by applying forces and torques directly.

3.5.1. External Applied Torques

You can apply a pure *external* torque to cause rotation of the body about an axis by giving a 3d torque vector which is added to the net torque vector τ used in the dynamics equations for rotation.

3.5.2. External Applied Forces

Forces require both a 3d vector for the force itself and a 3d vector for its point of application. If is often most convenient to specify the force in terms of worldspace coordinates (converting it to the coordinates of the local frame of the segment upon which it is acting before doing the dynamics equations). The force itself is added to the net force used in the translational equations of motion **f**.

The position of the force is essential because the force may also cause a torque, depending upon where it is applied. It is usually most convenient to specify the torque in terms of the local coordinate frame, e.g., pick a local point of application **p**. The torque due to the force is found by Equation 16.

3.6. Internal Control

Internal control is mostly relevant to moving an articulated body in the way robots and animals move themselves, by applying torques and forces between neighboring segments. As the dynamics formulation described for articulated bodies only accommodates rotary joints, only internal torques, not forces will be mentioned.

3.6.1. Internal Torques

If you would like the torque to be *internal*, e.g., simulating a muscle that acts upon two neighboring segments in an equal and opposite fashion, this torque should contribute to the net torque on one segment and its negative should contribute to the net torque on its neighbor.

Internal torques are also useful for simulating joint limits, e.g., to keep the arm from bending backwards at the elbow. Rotary spring and damper combinations or exponential torques can be used to simulate them.

3.6.2. Positional Suggestions

Moving bodies about by suggesting forces and torques is less than intuitive. We usually think about motion kinematically, as changes in position. It is still possible to take advantage of dynamics but have the user think in positional terms by providing a (more or less) intelligent preprocessing step that converts positional suggestions to forces and torques that will accomplish them.

3.6.3. Internal Positional Control

The user could suggest local positional changes at joints, e.g., rotate the elbow from 45 degrees to 60 degrees in 10 seconds. The system could take into account the mass of the segments moving and their present velocity and guess how much internal torque will do this. Using super- or adaptive sampling or feedback, reasonable torques can be found to accomplish the desired motion. Before you ask why use dynamics at all, consider that only a few joints of the body need be under positional control at any time. The rest may be left in a simple state that is automatically dealt with, e.g., relaxed and hanging loosely, or frozen into a local configuration.

3.6.4. External Positional Control: Goals

It is sometimes handy to pick a point on a body and then a point in worldspace where you would like that point to be (a goal). In this case, you can apply a force starting at the desired body point and directed toward the goal. Finding the amount of force to pull the body to the goal at a reasonable speed without overshooting it or oscillating is sometimes tricky.

3.7. Environment Interactions

It would be nice if bodies could react automatically and realistically to their environment as well. This will add to the cost of the system, because considerable collision detection may have to be done. A simple brute force method of finding collisions is to check for the intersection of all the bounding vertices of an object with the bounding

planes of all other objects.

3.7.1. Floors

Floors can be simulated with reasonable success by modeling them as a combination of a spring and a damper. A spring supplies a force dependent upon the amount its compressed, δc, times a constant k.

$$f_{spr} = k\,\delta c \qquad 30.$$

Similarly, a damper supplies a force dependent upon its velocity times a constant.

For complex articulated bodies, it may be well not to use a constant constant for these equations, but find some way of automatically calculating a reasonable proportionality constant for the body considering its total motion.

3.7.2. Other Collisions

Collisions with other objects is not fundamentally different, though their shapes may be different and they may be expected to move in response as well. In this case, the collision should be recognized and the collisions forces found before dynamics is done on the individual objects to find their motion in response to the collisions. For simple bodies, one might prefer to calculate the effects of collisions directly, rather than simulating them with springs and dampers.

4. Numerical Issues

Dynamics is alot more expensive than kinematics, but not unreasonably so, given the rapidly decreasing cost of compute power. I imagine we could be doing this on modern personal computers without too much trouble; at least, if I had a modern personal computer, I'd try it. The bells and whistles are costly, e.g. collision detection, joint limits, internal preprocessed control, etc. Lots of work remains to be done on this. Use of recursive dynamic formulations is a real boon. More sophisticated numerical integration methods can also help, Runge-Kutte integration is somewhat more complex to program and takes longer per time step but you can use much larger time steps than with the Euler method and get more accurate results. Adaptive calculations can also help, e.g. use large time steps when the body is falling freely but very small ones when it hits the floor. A clever adaptive idea (thanks to Ralph Abraham, UCSC) is to do a 5th order and a 4th order Runge-Kutte integration and if they deviate more than some allowed amount, redo it with a smaller time step.

5. Who is Doing It?

This is by no means a complete list, but the people and places that I have heard are doing this sort of thing include: me (at UC Santa Cruz), Dave Forsey (at the University of Waterloo),[24] Bill Armstrong and Mark Green (at the University of Alberta),[1,3,2] Michael Girard and A. Maciejewski[9] (at the Ohio State University). Work being presented at SIGGRAPH '87 relating to this topic includes that of Haumann at Ohio State[11] Al Barr

and others (CalTech and elsewhere),[4] and Terzopoulos et al,[20] Isaacs and Cohen[13] and Witkin et al.[27]

6. Summary

This paper is a summary of the knowledge of dynamics that I've found useful for simulating the motion of bodies for computer animation. It's been an interesting and enjoyable way of creating animation, and seems to have a future. I hope you have fun with it and tell me if you have any problems or come up with any new solutions. Good luck!

Table 1. Meaning of Terms	
Matrices	
$\mathbf{J}$	= *inertial tensor matrix*
$\mathbf{R}^{topar}$	= *rotation matrix segment to parent*
$\mathbf{R}^{frompar}$	= *rotation matrix parent to segment*
$\mathbf{R}^{toworld}$	= *rotation matrix segment to world*
$\mathbf{R}^{fromworld}$	= *rotation matrix world to segment*
$\mathbf{D}$	= *rotation matrix seen as direction cosines*
$\mathbf{I}$	= *identity matrix*
$\mathbf{K}$	= *recursive coefficient matrix*
$\mathbf{M}$	= *recursive coefficient matrix*
3d Vectors	
$\mathbf{f}$	= *force*
$\mathbf{f}_{grv}$	= *force due to gravity*
$\mathbf{f}_{ext}$	= *external applied force*
$\mathbf{f}_{son}$	= *force applied by child of a segment thru a joint*
$\mathbf{f}_{topar}$	= *force applied onto parent of a segment thru a joint*
τ	= *torque*
τ_{grv}	= *torque due to gravity*
τ_{ext}	= *external applied torque*
τ_{son}	= *torque applied by child of a segment thru a joint*
τ_{topar}	= *torque applied onto parent of a segment thru a joint*
$\mathbf{p}$	= *position*
$\mathbf{v}$	= *linear velocity*
$\mathbf{a}$	= *linear acceleration*
$\mathbf{a}_{grv}$	= *gravitational acceleration*
$\delta\mathbf{u}$	= *angular position*
ω	= *angular velocity*
$\dot{\omega}$	= *angular acceleration*
$\mathbf{l}$	= *vector to joint of son segment from parent frame*
$\mathbf{c}$	= *vector to segment center of mass defined in segment frame*
$\mathbf{d}$	= *recursive coefficient*
$\mathbf{f}'$	= *recursive coefficient*
Scalars	
m	= *mass*
δt	= *time step between samples*
I_x, I_y, I_z	= *moments of inertia*
I_{xy}, I_{xz}, I_{yz}	= *products of inertia*

References

1. William W. Armstrong, "Recursive Solution to the Equations of Motion of an N-link Manipulator," *Proceedings Fifth World Congress on the Theory of Machines and Mechanisms*, pp. 1343-1346, Am. Soc. of Mech. Eng., 1979.
2. William W. Armstrong, Mark Green, and R. Lake, *Proceedings of Graphics Interface 86*, pp. 147-151, May, 1986.
3. William W. Armstrong and Mark W. Green, "The Dynamics of Articulated Rigid Bodies for Purposes of Animation," *Proceedings of Graphics Interface '85*, pp. 407-415, Computer Graphics Society, May, 1985.
4. Alan H. Barr, "Dynamic Constraints," *SIGGRAPH '87 Tutorial Notes: Topics in Physically-Based Modeling*, 1987.
5. S. Conte and C. de Boor, in *Elementary Numerical Analysis, 3rd edition*, McGraw-Hill Book Company, New York, 1980.
6. R. Featherstone, "The Calculation of Robot Dynamics Using Articulated-Body Inertias," *International Journal of Robotics Research*, vol. 2, no. 1, pp. 13-30, Spring, 1983.
7. Richard P. Feynman, Robert B. Leighton, and Matthew Sands, in *The Feynman Lectures on Physics*, California Institute of Technology, Pasadena, California, 1963 .
8. Daniel T. Finkbeiner, II, *Introduction to Matrices and Linear Transformations,* p. 174, W. H. Freeman and Company, San Francisco, CA, 1960. matrices
9. Michael Girard and Antony A. Maciejewski, "Computational Modeling for the Computer Animation of Legged Figures," *SIGGRAPH '85 Conference Proceedings*, vol. 19, pp. 263-270, July, 1985.
10. Donald T. Greenwood, in *Principles of Dynamics*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey.
11. David Haumann, "Modeling Flexible Bodies," *SIGGRAPH 1987 Tutorial Notes: Topics in Physically-Based Modelling*, July, 1987.
12. Roberto Horowitz, "Model Reference Adaptive Control of Mechanical Manipulators," PhD Thesis, Mechanical Engineering, University of California, Berkeley, California, May, 1983.
13. Paul M. Isaacs and Michael F. Cohen, "Controlling Dynamic Simulation with Kinematic Constraints," *SIGGRAPH 1987*, July, 1987.
14. C. S. George Lee, R. C. Gonzalez, and K. S. Fu, *Tutorial on Robotics,* IEEE Computer Society Press, Silver Spring, MD, 1983.
15. W. G. McLean and E. W. Nelson, *Engineering Mechanics: Statics and Dynamics,* Shaum's Outline Series, McGraw-Hill Book Co., New York, 1978.
16. NASA, *Anthropometric Source Book,* NASA Scientific and Technical Information Office, 1978.
17. L.A. Pars, *A Treatise on Analytical Dynamics,* Ox Bow Press, Woodbridge, Connecticut, 1979.
18. Richard P. Paul, *Robot Manipulators: Mathematics, Programming, and Control,* The MIT Press, Cambridge, MA, 1981.

19. Robert Resnick and David Halliday, *Physics Part I,* John Wiley and Sons, Inc., New York, 1966.

20. Demetri Terzopoulous, John Platt, Alan H. Barr, and Kurt Fleischer, "Elastically Deformable Models," *SIGGRAPH 1987*, July, 1987.

21. Dare A. Wells, *Lagrangian Dynamics,* Shaum's Outline Series, McGraw-Hill Book Co., New York, 1969.

22. Jane Wilhelms, "Virya - A Motion Control Editor for Kinematic and Dynamic Animation," *Proceedings of Graphics Interface 86*, pp. 141-146, May, 1986.

23. Jane Wilhelms, "Using Dynamic Analysis for Animation of Articulated Bodies," *IEEE Computer Graphics and Applications*, vol. 7, no. 6, June, 1987.

24. Jane Wilhelms, David Forsey, and Pat Hanrahan, *Manikin: Dynamic Analysis for Articulated Body Manipulation,* Computer and Information Sciences Board, U. of California, Santa Cruz, CA 95064, April, 1987. Tech. Report UCSC-CRL-87-2

25. Jane Wilhelms, "Graphical Simulation of the Motion of Articulated Bodies such as Humans and Robots, with Particular Emphasis on the Use of Dynamic Analysis," *PhD Thesis*, Computer Science Division, Berkeley, CA, July, 1985.

26. Jane Wilhelms and Brian A. Barsky, "Using Dynamic Analysis for the Animation of Articulated Bodies such as Humans and Robots," *Proceedings of Graphics Interface '85*, pp. 97-104, May 1985.

27. Andrew Witkin, Kurt Fleischer, and Alan H. Barr, "Energy Constraints on Parameterized Models," *SIGGRAPH 1987*, July, 1987.

The BRL CAD Package
An Overview

Phillip C. Dykstra

Advanced Computer Systems Team
U. S. Army Ballistic Research Laboratory
Aberdeen Proving Ground
Maryland 21005-5066 USA

ABSTRACT

The major components of the BRL CAD Package are reviewed. The BRL CAD Package is a combinatorial solid geometry (CSG) based modeling system which includes an interactive model editor, a ray tracing library, a generic framebuffer library, and a large collection of related tools.

An object-oriented ray tracing library provides the primary method of model interrogation. A whole family of engineering analysis applications based on the ray tracing paradigm has been built, including traditional renderers, and predictive radar models. A generic framebuffer library interface with transparent networking capability provides hardware independent access to any display device from any host. Several categories of software tools for image display, manipulation, and analysis are discussed. Some general user interface issues are mentioned.

This paper emphasizes the reasons which led to the system as is exists today, and comments on some of its various strengths and weaknesses.

1. Introduction

The Ballistic Research Laboratory CAD Package is a large body of software consisting mainly of 1) a solid model editor (MGED), 2) a ray tracing library for model interrogation (librt), 3) a generic framebuffer library with full network display capability (libfb), and 4) a large collection of software tools for framebuffer and image manipulation and analysis. Parts of this system have roots in work done over two decades ago, most notably the solid modeling, and the ray tracing. Recently this software has been through a new generation of growth. It is now distributed free of charge to many sites around the world on a non-redistribution basis.

As with many large systems, parts of it were the result of years of evolution, with many band-aids, hacks, and "backward compatibility" requirements along the way. The work that one needed to accomplish today was often more influential than any carefully made plans. Most of this history is known only to those who watched it happen.

This paper provides a brief overview of the major components of the BRL CAD system. It will attempt to explain how and why many parts of it are the way they are. Finally, it will entertain the question of what is good and bad about it, and how the various decisions that were made have or have not worked.

2. Solid Modeling - MGED

The BRL has been building solid models of vehicles and other objects for over twenty years. These models are analysed for various physical properties (such as center of mass, moments of inertia), vulnerability, and more recently for optical, radar, and IR signatures.

This work began in the early 1960s when BRL had the Mathematical Applications Group Inc. (MAGI) develop a method of geometric description for military vehicles.[1] The method decided upon was Combinatorial Solid Geometry (CSG). This is a system where various geometric solids (boxes, cones, ellipsoids, tori, etc.) are combined using boolean operations (union, intersection, and subtraction). CSG represents one of the two major classes of modeling, the other being surface or boundary representations (B-reps). A key reason for the selection of CSG modeling is that it is "true to reality." Physical objects are solids, not just surfaces. If an object has been constructed with CSG, one is at least assured of its physical possibility.

For several years, models were constructed on large sets of punch cards. One or more cards would contain the parameters for a particular solid; other cards would describe the boolean relationships between solids. This system was not hierachical, all solids and combinations existed at one level. Ray tracing was used to analyse these models, but the only images of these models ever produced were crude plotter drawn wireframes.

A new generation of modeling tools emerged in 1979-1980. A system was built which allowed these models to be interactively displayed and edited on vector display devices. The success of these early efforts, coupled with the failure to find commercial tools of sufficient power, led to the development of the MGED model editor. The MGED editor is written in C and has been run on a large variety of machines. An object oriented interface to a set of display managers allows many different display devices to be supported. The types of primatives supported include: arbitrary boxes of up to eight verticies, ellipsoids, truncated general cones, tori, polygonal solids, and solids constructed of B-spline surfaces.[2]

The CSG representation is a natural form for our most common method of model interrogation - ray tracing. There are some methods of analysis however for which a surface facet representation of a model is the desired form. Work is currently under way on the facetization of CSG models, in order to support the needs of such codes. Future work is also planned in automatic mesh generation for similar reasons. These two capabilities will further ease the barrier between model representation, and model analysis.

For a much more comprehensive coverage of solid modeling, with MGED as a case study, see Muuss.[3]

3. Model Analysis - Ray Tracing

Ray tracing is a method of point sampling a geometric model by mathematically intersecting lines with objects in the model. At each intersection point various properties of the model can be determined: where did it intersect, what is the surface normal and curvature at that point, what part of the model was hit, what are the material properties at that point, etc. The computer graphics community often cites the origins of ray tracing with Kay's 1979 thesis,[4] or Whitted's paper of 1980.[5] However, the use of ray tracing as a method of geometric model interrogation has its origin in a BRL contract with MAGI, the initial results of which were published in 1967.[1] More details on the origins of ray tracing can be found in Muuss.[6] For an overview of the method itself, see Rogers.[7]

Ray tracing is the primary method used by BRL for model interrogation. Many people in the computer graphics community dislike ray tracing, primarily due to its notoriously high computational expense compared to other rendering techniques. But there are several key reasons why BRL uses it: 1) We are primarily concerned with doing an engineering analysis of the model, not just making pretty pictures of it, this objective is what led us to CSG models to begin with. 2) When CSG models are used, ray tracing is the most common method for evaluating the boolean expressions, 3) Firing a ray at a model is very much like firing a projectile (or light) at it, and is thus a natural method for vulnerability and signature analysis.

The ability to intersect rays with a model is common to all of the analysis tools, whether one is rendering a picture of the model or computing a moment of inertia. For this reason, the code which knows how to efficiently trace rays through a CSG model has been put in a library, librt. An application linked to this library has complete control over which rays are fired, how much information is computed at the intersection points, and what is done with the returned information. This library level separation of ray tracing and analysis has proven to be an extremely good one.

Other splits between ray tracing and analysis have been made or proposed. Some systems trace the entire model, placing the results into an intermediate file. There are two problems with this: the analysis code can not influence the ray trace (for example, by deciding when to reflect or when to fire extra rays in an area), and the volume of data generated is extremely large, often filling an entire large disk drive. The split could also be implemented by passing messages between separate processes via a remote procedure call, or a stream mechanism such as a UNIX pipe. The amount of overhead involved with either of these methods is typically of the same order of magnitude as the work involved in tracing a single ray. This approach is thus felt to be impractical.

Two ray tracing programs which use librt are provided in the CAD package: RT and LGT. LGT is an optical rendering program with a *curses* based screen oriented user interface. RT also provides rendered images with command line arguments, but is itself the front end for several applications including a radar model. RT also has the ability to read scripts of commands which can control the computation of a sequence of frames, and the orientations and properties of materials in each frame of an animation.

Future work with the ray tracer includes extending the classes of traceable objects, further efficiency improvements, and its extension to handle a broader class of physical phenomena. The latter goal includes multiple spectral point sampling (instead of just Red Green Blue) to account for dispersion and complex spectra, divergence factors (for the concentration and diffusion of light), and polarization effects.

4. The Framebuffer Library

The framebuffer library (libfb) provides a device independent interface to a raster display. A program compiled with this library can access many different display types, including those on other machines on the network. The most important routines are summarized below.

libfb routines	
fb_open(device,width,height)	open the device
fb_close(fbp)	close the device
fb_read(fbp,x,y,buf,count)	read count pixels at x,y
fb_write(fbp,x,y,buf,count)	write count pixels at x,y
fb_clear(fbp,color)	clear to an optional color
fb_rmap(fbp,colormap)	read a colormap
fb_wmap(fbp,colormap)	write a colormap
fb_window(fbp,x,y)	place x,y at center
fb_zoom(fbp,xzoom,yzoom)	pixel replicate zoom
fb_getwidth(fbp)	actual device width in pixels
fb_getheight(fbp)	actual device height
fb_cursor(fbp,mode,x,y)	cursor in image coords
fb_scursor(fbp,mode,x,y)	cursor in screen coords
fb_log(format,arg,...)	user replaceable error logger

The coordinate system for x,y specifications is first quadrant. While we went round and round about first vs. fourth quadrant with arguments akin to "which end of the egg first", the decision for first quadrant resulted primarily because that is the same ordering as our image files (.pix files, see below). The image files themselves were ordered that way because Utah's RLE files are first quadrant. If reads and writes extend beyond the end of a scanline, they wrap in first quadrant fashion.

The pixels passed to and from the library are simply arrays of bytes interpreted as RGBRGB.... While we used to define a pixel structure with red, green, and blue elements, this was changed to a typedef'd array of three unsigned chars. This was important in order to avoid structure padding. The Cray computers for example would have used eight bytes per pixel with the old format. Unfortunately, one does run into some compiler touchiness when using pointers to typedefs which are themselves arrays!

The display to be used is selected by a command line argument, an environment variable FB_FILE, or a default for the system the code is running on. The format is [host:]/dev/*device_name*[#], or simply "filename". The /dev/ part is used to identify a display device. The *device_name* need not correspond to entries in /dev, it is just that if the /dev prefix is not given a file pathname is assumed. If a hostname is given, a network connection is opened to the framebuffer library daemon (rfbd) on that machine. The remaining part of the string is passed to that host for the open (this generalizes the open to allow multiple "hops" in order to get to a host). Currently supported displays include the Adage Ikonas, Silicon Graphics Iris, black and white and color Sun workstations, and AT&T 5620 terminals. There is also a debug interface, and a disk file interface.

A set of buffered I/O routines is also provided. In this interface a "band" of scanlines is kept in memory and the appropriate pre-reads and flushing is done. While this interface can speed up single pixel reads and writes, it does not make the drawing of vertical lines any easier, since such a line would run through several bands. In practice, very few of our programs use buffered I/O. Most programs keep their own scanline buffers and do unbuffered scanline size reads and writes. Some thought has been given toward allowing the selection of the memory buffering mode at run time, perhaps keyed on a device name parameter. This would permit the user to control the trade off between speed and interactive output. The ability to make such a decision becomes particularly important when one is using a remote display.

libfb buffered I/O	
fb_ioinit(fbp)	set up a memory buffer
fb_seek(fbp,x,y)	move to an x,y location
fb_tell(fbp,xp,yp)	gives the current location
fb_rpixel(fbp,pixelp)	read a pixel and bump location
fb_wpixel(fbp,pixelp)	write and bump current location
fb_flush(fbp)	bring display up to date

The framebuffer library owes much of its current form to its history. One of the first true framebuffers purchased by BRL was an Ikonas (now Adage RDS-3000), in 1981. This device runs as either a 512x512 or 1024x1024 display with 24 bit pixels. It has three 256 entry 30-bit (10 bits per DAC) colormaps, hardware pan and zoom, and hardware cursor support. Michael Muuss of BRL wrote our first library for that device (libik).

Later, a Raster Technologies One/180 framebuffer was acquired and a libik like interface was created for it. As other devices followed, libfb was born. At first there was a switch in every library routine for every display device. Later it was reworked to have an object oriented interface: opening a device fills in a function switch table with that display's routines, and a "framebuffer pointer" was returned to that structure. Most of the framebuffer routines became macros which vector directly out to the device dependent code.

Finally, the machine which had our nice displays on it (a VAX 11/780) was also one of our slowest. To make this less of an issue, a libfb look alike was put together one evening which passed all library calls and returns across a network connection to a daemon that made calls to a "real" libfb. This was facilitated by the Package Protocol[8] (PKG) which allows messages to be exchanged, both synchronously and asynchronously, across a TCP connection (this protocol had originally been developed to make a remote MGED display possible, but later found uses in command and control experiments, etc.). The remote framebuffer code was merged into libfb during its object-oriented restructuring, so that one need only link with a single library to get both local and remote display capability.

Starting with the Ikonas in some sense spoiled us. It gave us full color pixels, colormaps, cursors, and pan and zoom. These features were incorporated into the generic framebuffer model used in our library. This makes fitting devices like the Sun workstations into our library quite trying, but this difficulty is more the result of things that workstations like the Sun can't do than it is a design problem with our library. On the other hand, the Ikonas also left us with programs that have to open the device in one of two "modes", either high or low resolution. To make matters worse, it does not allow the current display mode to be read back from the hardware. Therefore, the open must set the Ikonas to a known state. As a result, every framebuffer program, even those which have little to do with display size (such as those which read or write colormaps), carries around a "hires" flag so the device can be opened in the proper "mode."

One commonly asked question is whether X Windows will make the BRL framebuffer library unnecessary. X currently cannot support 24-bit color images, nor does it provide a powerful enough interface for controlling many framebuffer operations (e.g. colormaps, pan and zoom). If these deficiencies are overcome than X may prove to be a suitable replacement for the framebuffer library. In the near term, an X based module implementing a subset of the framebuffer library functions will likely be developed.

5. The Software Tools

A large number of simple tools for manipulating images and framebuffers are provided in the CAD package. They have been written in the traditional UNIX Software Tools fashion: each performs a simple basic function, with a minimum of back talk, and is intended to be hooked together with other tools to achieve an overall goal. A fair amount of effort has gone into making a standard interface to the tools. All tools provide a usage message if executed with no arguments (often after checking for a tty on stdin or stdout when it expects binary data), and common collection of flags is defined for all of the tools.

The use of software tools for computer graphics is not new. Recent systems advocating this tools based approach include those of Duff[9] and Peterson.[10] The BRL CAD Package has proven to be extremely flexible as as result of this approach. Generally, a new tool is added whenever the existing ones are found to be inadequate. Success can be claimed if one can easily achieve day-to-day tasks without having to write specialized programs.

5.1. File and Image Formats

Several kinds of files are read and generated by programs in the CAD package. These include model databases in a binary form (with a typical filename extension of .g), portable ASCII versions of those (.asc), and University of Utah Run Length Encoded (RLE) images (.rle). By far the most common image format for the tools however is either eight bit per pixel black and white (.bw) or 24-bit per pixel color (.pix). The files have the simplest format imaginable: there is no header at all, and pixels run in first quadrant order - lower left corner, across the scan lines, bottom scan line first, up through the top scan line. The values in the bytes are viewed an intensities from 0 (off), through 255 (full on). The color (.pix) files are in RGBRGB... order. Note that while we use the University of Utah RLE format, we view it simply as a means of image compression, unlike Utah which actually manipulates RLE files directly in their Raster Toolkit.[10]

The use of a simplistic headerless image format is perhaps the most debatable decision we made. It's primary advantage comes when piping several tools together. Each program is simply handed data. It doesn't have to know "how" to read it; there is no header to discard, or harder still, it doesn't have to do the "right thing" with the header information. Doing the "right thing" is extremely complicated if the header contains very much information. We have also avoided the N^2 problem of format conversion by converting all other formats into and out of this simple one.

Having "raw" headerless data has its price however. It is difficult to tell whether a given image is color or not, what its dimensions are, etc. File naming conventions (.bw or .pix) solve the first; "standard sizes" of 512x512 or 1024x1024 (hires) help alleviate the second (recall that these came from the Ikonas framebuffer). Note that usually only the scanline length needs to be

known, the number of lines can then be found by the file size. Many algorithms simply run until all of the data is gone, and some don't even care about scanlines at all.

5.2. Format Conversion

Several other image formats are accommodated by "filters" that convert one into the other. A selection of these is listed in the table. In all of the tables given the reverse conversion is omitted, e.g. there is also a pix-rle for converting color images into RLE format. Also, only the color (pix) version of a tool has been shown while most have black and white (bw) equivalents. Most of the tools listed also allow a wide variety of options. The color to black and white converter for example (pix-bw), allows either equal, NTSC, or "typical" CRT weighting to be applied. It also allows arbitrary weights to be given for selecting or mixing of the color planes in any way desired.

Selected Format Conversion Tools	
g2asc	model database to portable ascii form
bw-pix	black and white to color image
bw3-pix	three black and whites to color RGB
rle-pix	Utah's RLE format to color image
ap-pix	Applicon Ink-Jet to color image
sun-pix	Sun bitmap to color or black and white
mac-pix	MacIntosh MacPaint bitmaps to color

5.3. Framebuffer Tools

We have chosen to do most of the image manipulation and processing either on data streams, or on disk files. This was done in order to separate the notion of a device from image handling. A common beginning or end of a processing pipeline is to get or put an image into or from a framebuffer. Framebuffers do allow one to manipulate images in many useful ways however, so some device independent tools are provided for that. These include tools to allow changing colormaps, panning and zooming through an image, labeling, etc. Where tools require the user to move a cursor or the image, both EMACS and VI style commands are accepted by all programs.

Selected Framebuffer Tools	
fb-pix	framebuffer to color image
fb-bw	framebuffer to black and white
fb-cmap	read a framebuffer colormap
fbcmap	can load several "standard" colormaps
fbclear	clear to an optional RGB color
fbgamma	load or apply gamma correcting colormaps
fbzoom	general zoom and pan routine
fbpoint	select pixel coordinates
fblabel	put a label on an image
fbcolor	a color selecting tool
fbscanplot	scanline RGB intensity plotter
fbanim	a "postage-stamp" animator
fbcmrot	a colormap rotator
fbed	a framebuffer image editor

5.4. Image Manipulation

A collection of tools for image manipulation are provided. These can generate statistics, histograms, extract parts of an image, rotate, scale, and filter them, etc. Some of these are listed in the table.

Selected Image Tools	
pixstat	statistics - min, max, mean, etc.
pixhist	histogram
pixhist3d	RGB color space cube histogram
pixfilter	apply selected 3x3 filters
pixrect	extract a rectangle
pixrot	rotate, reverse, or invert
pixscale	scale up or down
pixdiff	compare two images
pixmerge	merge two/three images
pixtile	mosaic images together
gencolor	source a byte pattern
bwmod	apply expressions to each byte

6. User Interface

Using software tools effectively comes with experience. The BRL CAD Package has tried to ease the difficulty of learning a new set of tools by using a common set of flags and common tool naming conventions throughout the package. The "user interface" is ultimately the Unix shell, and its conventions for establishing pipes, passing arguments to programs, etc. A shell with history recall and editing, such as the *tcsh*, is almost a necessity when constructing complicated command line pipes.

Constructing complex interconnections between processing tools from the command line is sometimes difficult. One limitation is the single input single output notion of a Unix pipe. Image manipulation often calls for three or more channels of data. The most common solution to this problem is the use of intermediate files. Other approaches include extensions to the *tee* program, or a special tool such as *chan*[11] which demutiplexes a stream, feeds each channel to a different program, and remultiplexes the results.

Recently several systems have been developed to facilitate the coupling of dataflow oriented tools. Stephen Willson of NRTC has developed what he calls a Layered User Interface.[12] This is a set of tools that provides generic buttons and sliders which can pass values on as tool arguments. Several of the BRL CAD tools have been used in this environment. Dave Tristram of NASA Ames has put together a system called Flowtools[13] which allows the connections between tools to be specified with a dataflow like language, including inputs from sliders, etc. Both of these systems allow complex custom applications to be put together without writing any code.

7. Conclusions

The BRL CAD Package is a Unix based system which provides a CSG solid model editor, a ray tracing library for model interrogation, a generic framebuffer library with network display capability, and a large collection of software tools. The library level interface to the ray tracer has allowed a large collection of model analysis tools to be incorporated into the system. The generic network capable framebuffer library has proven to be of tremendous day to day importance.

The package provides a flexable set of software tools for image manipulation. The image formats are extremely simplistic, something which has proven to have both good and bad characteristics. Approaches to providing higher level interfaces to tools of this form have been indicated.

1. MAGI Inc, *A Geometric Description Technique Suitable for Computer Analysis of Both Nuclear and Conventional Vulnerability of Armored Military Vehicles,* MAGI Report 6701, AD847576 (August 1967).
2. P. R. Stay, "The Definition and Raytracing of B-spline Objects in a Combinatorial Solid Geometric Modeling System," *USENIX: Proceeding of the Fourth Computer Graphics Workshop* (Oct 1987).
3. M. J. Muuss, "Understanding the Preparation and Analysis of Solid Models," in *Techniques*

for Computer Graphics, ed. D. A. Rogers, R. A. Earnshaw, Springer-Verlag (1987).

4. D. S. Kay, *Transparency, Refraction, and Ray Tracing for Computer Synthesized Images,* Cornell Univ (Jan 1979).
5. J. T. Whitted, "An Improved Illumination Model for Shaded Display," *Communications of the ACM* **23**(6), pp. 343-349 (June 1980).
6. M. J. Muuss, "RT and REMRT - Shared Memory Parallel and Network Distributed Ray-Tracing Programs," *USENIX: Proceeding of the Fourth Computer Graphics Workshop* (Oct 1987).
7. D. F. Rogers, *Procedural Elements for Computer Graphics,* McGraw-Hill, New York (1985).
8. M. J. Muuss, P. Dykstra, K. Applin, G. Moss, E. Davisson, P. Stay, C. Kennedy, *Ballistic Research Laboratory CAD Package, Release 1.21,* BRL Internal Publication (June 1987).
9. Tom Duff, "Compositing 3-D Rendered Images," *Computer Graphics 19(2):41 (Proceedings of SIGGRAPH 85)* (July, 1985).
10. J. W. Peterson, R. G. Bogart, and S. W. Thomas, "The Utah Raster Toolkit," *USENIX: Proceeding of the Third Computer Graphics Workshop* (1986).
11. R. F. Moore, *CARL Startup Kit,* Computer Audio Research Laboratory, UCSD (1985).
12. S. Willson, "The Layered User Interface," *IRIS Universe* (To appear, Fall 1987).
13. David Tristram, "FlowTools: Dataflow Graphics Under Unix," *to appear, IEEE Conference on Workstations*, NASA Ames Research Center.

The Definition and Ray-tracing of B-spline Objects in a Combinatorial Solid Geometric Modeling System

Paul Randal Stay

Advance Computer Systems Team
US Army Ballistic Research Laboratory
Aberdeen Proving Ground
Maryland 21005-5066 USA

ABSTRACT

Traditionally there has been a distinction between Combinatorial Solid Geometry (CSG) modeling systems and Sculptured Surface Design modeling systems. CSG modeling systems largely model parts which are unsculptured and consist of combinations of common shapes like spheres, prisms, ellipsoids, and the like. These shapes are represented as planar half spaces, and algebraic quadratic surfaces. The boolean combination of these surfaces is usually performed by ray-tracing. Sculptured Surface Design concerns itself with modeling the surface of an object, i.e., the boundaries of an object like an aircraft, a ship, or an automobile. The boundaries are represented by using parametric tensor-product surfaces consisting of Bezier curves and Nonuniform Rational B-spline Surfaces (NURBS). There are many times however, when both modeling approaches are needed. In particular it is often desirable to introduce free-form surfaces into the CSG system. Recent advances in ray-tracing free-form surfaces have allowed the integration of free-form objects in CSG systems. This presentation will discuss the development and integration of NonUniform Rational B-splines into the BRL CSG modeling system.

1. Introduction.

Computer Aided Geometric Design, since its beginning in the mid-1960's has taken two different approaches to the modeling of mechanical parts and objects: sculptured surface design and combinatorial solid geometry (also known as volumetric solid modeling). Each approach was developed to represent different types of objects and requires a different style of object definition. Sculptured surface design concerns itself with modeling the surface boundaries (i.e., an aircraft or ship hull). CSG systems model parts that are unsculptured and consist of combinations of common shapes like spheres, prisms, cones, and the like.

Free-form surfaces are hard to represent as boolean combinations of these volumetric solids, therefore a faceted polyhedron was introduced to allow for a rough approximation of the surface geometry. Faceted polyhedra are useful in many applications and analyses that require a minimum amount of surface geometry. However, geometric information such as gaussian curvature requires a more accurate description of the surface geometry than is currently available using a faceted approach. There are many problems associated with the extension of CSG modeling systems to include sculptured surface primitives. The addition of these free-form surfaces, independent of their representation, requires the introduction of a new surface object type to the CSG system. These free-form object types can be defined by using either an implicit mathematical representation (e.g. superquadrics) or a boundary representation. In the case of "superquadrics" the modeling system does not inherit a well developed sculptured surface form. The same style of inside and

outside determination cannot readily be made with the boundary model as with the procedural solid primitive representations, hence problems arise in implementing the boolean combinations of the solids made with sculptured surfaces.

2. Techniques for Rendering Boolean combinations of surfaces.

One approach, applied by the University of Utah Alpha_1 project is to treat all objects as sculptured surfaces represented as tensor product Nonuniform Rational B-spline Surfaces (NURBS).[1] Solid volumes are represented by a collection of B-spline surfaces called a shell. A key ingredient of the Alpha_1 system is the Oslo algorithm[2] which provides a computational technique for subdividing B-spline surfaces. Using the algorithm, Spencer Thomas [3] has defined and implemented a classification scheme that allows boolean operations to be performed on sculptured surfaces. An interesting sidelight of this intersection algorithm is that B-spline surfaces do not need to describe a closed volume, allowing for the ability to have partially bounded sets.

Since NURBS are the fundamental representation, each of the CSG solids can be derived and represented with other defined volumetric primitives such as rounded edge boxes made of collections of B-spline surfaces.[4,5] There are two drawbacks to this approach however. First representing a sphere as a NURB may not be as efficient as a CSG representation. Secondly the representation of the intersection of two B-spline surfaces is not a B-spline surface but a collection of polygons.

Another approach, used by the Reyes image rendering system,[6] involves an extended Z-buffer algorithm which stores multiple z values for each solid to allow boolean operations between objects.

For the past 20 years, the Ballistic Research Laboratory has been using boolean combinations of simple volumetric shapes to design and analyze US Army vehicles.[7] Ray/solid intersection algorithms generate line segments that are used to classify the solids for boolean combinations. New advances in ray-tracing [8,9] show it is possible to calculate ray intersections with tensor product NURBS. This gives the capability to represent free-form surfaces in a CSG modeling system with additional surface geometric information and allows boolean combinations between solids in the system. Since ray-tracing is required for many of the applications within the BRL CSG modeling system, the ray/B-spline intersection algorithm has been integrated into that system.

3. B-spline Solid Definitions.

Tensor product Nonuniform Rational B-spline Surface properties have been discussed in a number of papers [10,11,12] and there have been a number of modelers written to edit splines. No attempt will be made here to discuss the different approaches in modeling systems that are used to create and manipulate NURBS.

A B-Spline solid can be defined as a collection of tensor product Nonuniform Rational B-spline Surfaces. These surfaces are used to define the boundary of the volume which is to be represented. However, there are constraints that must be met if NURBS or any other boundary representation is to be integrated into a CSG system.

Since all ray/solid intersections are required to perform boolean operations between surfaces, each surface or collection of surfaces must completely enclose space. Surfaces which are joined need to be specified such that the common boundary curves exactly match and that no gaps exist. This is required to ensure that the primitive represents a solid.

Surface normals of the NURB solid are required to point outward. This guarantees that the boolean operations and applications (such as rendering) result in solids which are consistent within the CSG system.

4. Ray-trace Algorithm

There have been many different approaches proposed for the ray/B-spline intersection algorithm. The most notable of these use either Newton's iteration method for determining the intersection point,[8,9] or tessellate the surface into a polygonal mesh. [13] Techniques which use the Newton iteration method tend to be computationally intensive, but do not lose the topology of the B-

spline surface. While less computationally intensive than the Newton iteration method, techniques that subdivide the surface into polygons tend to lose the topology of the B-spline surfaces.

The ray/B-spline solid intersection routines used in the BRL CAD ray-trace library[14] are based on techniques that were outlined in the original Oslo algorithm paper. Since the B-spline surface lies within the convex hull of the control mesh, a bounding box of the surface can be described by taking the minimum and the maximum of the de Boor net. The subdivision is performed by adding order multiple knots at the parametric midpoint in one of the given directions. The result of the subdivision is two distinct B-spline surfaces that represent the original surface. The extent of subdivision is determined by the following conditions, with more subdivision necessary if the condition(s), checked in the order listed, exist.

1. The ray intersects the bounding box of the convex hull boundary of the surface.
2. Interior knots of the B-spline exist.
3. The surface is not flat according to some flatness criteria.

Since ray-tracing is performed in object space, traditional scan line techniques for determining a flatness parameter for the surface are invalid. Flatness testing of the surface uses a modified form of cone and beam tracing. Each ray that is generated by the application program is given an initial beam radius r and a slope of beam divergence per millimeter s. One of the results of the ray/bounding box intersection is a parametric distance t from the ray origin to the bounding box of the surface. A variance parameter v is calculated by $v=r+st$ which is used to test the subdivided surface. Points from each row and column of the control mesh are then used to test for flatness. l_i is a line segment which is defined by two distinct points of the row/column, and d_j is the distance of each individual point to the line segment. If the condition $d \leq v$ is true, then the row is determined to be flat. If all rows and columns of the control mesh are flat then a further test is performed on the bezier points of the subdivided patch. A plane is formed from three of the surface corner points. If the distance from the fourth corner point to the plane is $\leq v$ then the surface is determined to be flat.

When a surface is determined to be flat, the four corner points of the control mesh are used to create two polygons which are then intersected with the ray.

If a ray intersects the bounding box of a B-spline surface, then the surfaces are recursively subdivided and tested against the ray until surface flatness criteria are reached or the ray misses the surface. The algorithm is as follows:

```
for each (surface in the B-spline Solid) add surface onto active node list
while (surfaces exist in the active list)
        Get first surface on the active list:
        if (the ray intersects the bounding box of the surface)
                If (the surface is flat)
                        intersect ray with the polygons
                        and sort hit point into the hit list.
                else (the surface is not flat)
                        subdivide the surface and insert the
                        two returned surfaces on the active list.
        else remove from active list.
continue until no surfaces exist on the active list
```

The sorted hit points are used to create line segments that describe the ray/solid intersection. All line segments are collected and the boolean operations are performed on all solid segments.

The B-spline surface subdivision tends to be computationally expensive to perform on conventional computers. However, the algorithm can be optimized by generating and storing the bounding boxes and the subdivided surfaces in a binary tree. The ray can then be tested recursively against the stored binary tree. Subdivision of the B-spline surfaces is performed at the time of the ray intersection testing, thus only those portions of the tree that were intersected by a ray are generated.

Many of the ray-tracing applications at the BRL need to calculate principle curvature in each direction on a surface. A method of calculating the derivatives of a B-spline surface using the control points[2] can be used for non rational B-splines.

5. Rational B-Spline surfaces.

Rational B-spline surfaces are used to exactly represent conic sections such as ellipsoids and hyperbolas and are important to aircraft designers. The rational B-spline is defined as:

$$S(u,v)=\left(\frac{x(u,v)}{\omega(u,v)}, \frac{y(u,v)}{\omega(u,v)}, \frac{z(u,v)}{\omega(u,v)}\right)$$

The Oslo algorithm can be applied to both the numerator as well as the denominator. The ω values are weights assigned to each of the points in the control mesh and can be represented in homogeneous space.[15] Rational surfaces which pass the flatness test divide the ω values of the corners from the control mesh to form the 3 space polygon points, which are passed to the ray/polygon intersection routine.

Rational surface must be treated separately for the calculation of curvature since the quotient rule must be applied. The formula for calculating the derivative of a rational B-spline in the u parametric direction is:

$$\frac{\partial S}{\partial u}=\frac{\omega(u,v)\left(\frac{\partial x}{\partial u}, \frac{\partial y}{\partial u}, \frac{\partial z}{\partial u}\right)-\left(x(u,v), y(u,v), z(u,v)\right)\frac{\partial \omega}{\partial u}}{\omega(u,v)^2}$$

There is hope that the computation can be made a bit more reasonable. Essentially one can still use the de Boor algorithm with the homogenous points in the control mesh. Substitution can then be used to calculate the derivative values. Thus, the $\frac{\partial S}{\partial u}$ can be expressed as follows:

$$\frac{\partial S}{\partial u}=\frac{1}{w(u,v)}\left(\frac{\partial x}{\partial u}, \frac{\partial y}{\partial u}, \frac{\partial z}{\partial u}\right)-\frac{\frac{\partial \omega}{\partial u}}{\omega(u,v)^2}\left(x(u,v), y(u,v), z(u,v)\right)$$

Similar expressions can be calculated for the rest of the derviatives for calculating the principle curvature.

6. Future Work.

Research in new computer hardware and software techniques should improve the speed of the ray/B-spline intersection.calculations.

In the hardware department, there are sections of the subdivision code that may take advantage of vectorization and parallelization such as on the Alliant and Cray computer systems. Specialized VLSI hardware is being developed by the University of Utah Alpha_1 project that will execute the Oslo algorithm and allow fast subdivision of Nonuniform Rational B-spline Surfaces. The

use of this specialized hardware will not only facilitate faster subdivision of the B-spline surfaces but will allow for some generality in the possible ray/bounding box intersection routines.

There are a number of software optimizations that will be investigated which may improve the algorithm. One area is that of the amount of memory which is necessary to store the binary tree and its subdivided surfaces. Currently the subdivision code returns two surfaces by performing the subdivision in the original surface. Refining the surface instead of splitting it will eliminate the excess data now common between the two surfaces returned from the subdivision algorithm. The routine to check for flatness should be able to return a direction for the subdivision since it can find the area of greatest variance in the control mesh. This will allow the surface to be subdivided in the direction of the larger parametric surface curvature.

7. Acknowledgments.

I would like to thank Peter Stiller who helped with the formulation of the derivatives for both rational and non rational B-splines surfaces. Thanks also go to the people of the Alpha_1 project at the University of Utah who have generated a rich set of B-spline based editing programs and tools. Thanks also to Paul Deitz and Mike Muuss for their influence and ideas.

1. E. Cohen, "Mathematical Tools for a Modelers Workbench," *IEEE Computer Graphics and Applications* (October 1983).
2. R. F. Riesenfeld, E. Cohen, T. Lyche and C. deBoor, "A Practical Guide to Splines," *Computer Graphics and Image Processing*, New York **14**(2), pp. 87-111, Springer-Verlag (1978).
3. S. W. Thomas, *Modelling Volumes Bounded by B-spline Surfaces,* PhD dissertation, University of Utah (June 1984).
4. P. R. Stay, *Rounded Edge Primitives and their Use in Computer Aided Geometric Design*, MS dissertation, University of Utah (August 1984).
5. Pat Hanrahan, "A Survey of Ray-Surface Intersection Algorithms," *SigGraph 87 Course Notes Introduction to Ray Tracing*, Anaheim, California **13** (July 27, 1987).
6. Cook, Carpenter, Catmull, "The Reyes Image rendering Architecture," *Computer Graphics (Proceedings of Siggraph '87)* **21**(4) (July 1987).
7. P. H. Deitz, *Solid Modeling at the US Army Ballistic Research Laboratory,* Proceedings of the 3rd NCGA Conference (13-16 June 1982).
8. M. A. J. Sweeny, R. H. Bartels, "Ray Tracing Free-Form B-spline Surfaces," *IEEE Computer Graphics* (February 1986).
9. John W. Peterson, "Ray Tracing General B-Splines," *Proceedings of the ACM Rocky Mountain Regional Conference*, p. 87 (April, 1986).
10. E. S. Cobb, *Design of Sculptured Surfaces using the B-spline Representation,* PhD dissertation, University of Utah (June 1984).
11. Russ Fish, "Alpha_1:modeling with nonuniform rational b-splines," *Iris Universe* **1**(1) (Spring 1987).
12. Bohm, Farin, Kahmann, "A survey of curve and surface methods in CAGD," *Computer Aided Geometric Desgin* **1**(1) (1984).
13. J. M. Snyder, A H. Barr, "Ray Tracing Complex Models Containing Surface Tessellations," *Computer Graphics (Proceedings of Siggraph '87)* **21**(4) (July 1987).
14. Michael John Muuss, "RT and REMRT shared Memory Parallel and Network Distributed Ray-Tracing Programs," *Proceedings of the Usenix Computer Graphics Workshop* (October 8-9, 1987).
15. W. Tiller, "Rational B-splines for curves and surface representation.," *IEEE Computer Graphics and Applications* **3**(6) (Sept. 1983).

RT & REMRT
Shared Memory Parallel
and
Network Distributed
Ray-Tracing Programs

Michael John Muuss

Leader, Advanced Computer Systems Team
U. S. Army Ballistic Research Laboratory
Aberdeen Proving Ground
Maryland 21005-5066 USA

ABSTRACT

The ray-tracing procedure is ideal for execution in parallel, both in tightly coupled shared-memory multiprocessors, as well as loosely coupled ensembles of computers. RT, the ray-tracer in the BRL CAD Package, takes advantage of both types of parallelism, using different mechanisms. The presentation will start with a discussion of the structure of the ray-tracer, and the strategies used for operating on shared-memory multiprocessors such as the Denelcor HEP, Alliant FX/8, and Cray X-MP.

The strategies used for dividing the work among network connected loosely coupled processors will be presented. This will include details of the dispatching algorithm, the distribution protocol designed, and a brief description of the "package" (PKG) protocol which carries the distribution protocol. The presentation will conclude by investigating the performance issues of this type of parallel processing, including a set of measured speeds on a variety of hardware.

1. Raytracing Background

The objective of a model analysis application determines the most natural form in which the model might be interrogated. For example, extracting just the *edges* of the objects in a model would be suitable for a program attempting to construct a wire-frame display of the model. Applications also exist which need to be able to find the intersection between the paths of small objects such as photons and the model. Interrogations such as these are motivated by a desire to simulate physical processes, and each alternative is useful for a whole family of applications.

Most physical objects have a significant cross-sectional area. Mathematical rays, however, have as their cross-section a point. Therefore, interrogating the model geometry with rays can result in sampling inaccuracies. While recent research has begun to explore techniques for intersecting cylinders, cones,[1,2] and planes with the model geometry,[3] ray-tracing is by far the most well developed approach. Fortunately, most applications can function well with approximate, sampled data. Data with statistical validity can be obtained by sampling the model with an adequate number of rays and computing the ray/geometry intersections. By choosing a ray sampling density within the Nyquist limit, these applications are satisfied by extracting ray/geometry intersection information, the well known "ray-tracing" algorithm. This approach is one of the easiest to implement, as the one-dimensional nature of a mathematical ray makes the intersection equations relatively straightforward, even with combinatorial solid geometry (CSG) models.

The origins of modern ray-tracing come from work at MAGI under contract to BRL, initiated in the early 1960s. The initial results were reported by MAGI[4] in 1967. Extensions to the early developments were undertaken by a DoD Joint Technical Coordinating Group effort, resulting in publications in 1970[5] and 1971.[6] A detailed presentation of the fundamental analysis and implementation of the ray-tracing algorithm can be found in these two documents. They form an excellent and thorough review of the principles of ray-tracing and solid modeling.

More recently, interest in ray-tracing developed in the academic community, with Kay's[7] thesis in 1979 being a notable early work. One of the central papers in the ray-tracing literature is the work of Whitted.[8] Model sampling techniques can be improved to provide substantially more realistic images by using the "Distributed Ray Tracing" strategy.[9] For an excellent, concise discussion of ray-tracing, consult pages 363-381 of Rogers.[10]

There are several implementation strategies for interrogating the model by computing ray/geometry intersections. The traditional approach has been batch-oriented, with the user defining a set of "viewing angles", turning loose a big batch job to compute all the ray intersections, and then post-processing all the ray data into some meaningful form. However, the major drawback of this approach is that the application has no dynamic control over ray paths, making another batch run necessary for each level of reflection, etc.

In order to be successful, applications need: (1) dynamic control of ray paths, to naturally implement reflection, refraction, and fragmentation into multiple subsidiary rays, and (2) the ability to fire rays in arbitrary directions from arbitrary points. Nearly all non-batch ray-tracing implementations have a specific closely coupled application (typically a model of illumination), which allows efficient and effective control of the ray paths. However, the most flexible approach is to implement the ray-tracing capability as a general-purpose library, to make the functionality available to any application as needed. This is the approach taken in the BRL CAD Package,[11] a large modeling and analysis system based primarily on the ray-tracing of CSG solid models. The ray-tracing library is called librt, while the ray-tracing application of interest here (an optical spectrum lighting model) is called RT. This software is available from the author at no charge on a non-redistribution basis.

2. The Structure of librt

In order to give all applications dynamic control over the ray paths, and to allow the rays to be fired in arbitrary directions from arbitrary points, BRL has implemented its second generation ray-tracing capability as a set of library routines. Librt exists to allow application programs to intersect rays with model geometry. There are four parts to the interface: three preparation routines and the actual ray-tracing routine. The first routine which must be called is rt_dirbuild(), which opens the database file, and builds the in-core database table of contents. The second routine to be called is rt_gettree(), which adds a database sub-tree to the active model space. rt_gettree() can be called multiple times to load different parts of the database into the active model space. The third routine is rt_prep(), which computes the space partitioning data structures and does other initialization chores. Calling this routine is optional, as it will be called by rt_shootray() if needed. rt_prep() is provided as a separate routine to allow independent timing of the preparation and ray-tracing phases of applications.

To compute the intersection of a ray with the geometry in the active model space, the application must call rt_shootray() once for each ray. Ray-path selection for perspective, reflection, refraction, etc, is entirely determined by the application program. The only parameter to the rt_shootray() is a librt "application" structure, which contains five major elements: the vector a_ray.r_pt ($\vec{P}$) which is the starting point of the ray to be fired, the vector a_ray.r_dir ($\vec{D}$) which is the unit-length direction vector of the ray, the pointer *a_hit() which is the address of an application-provided routine to call when the ray intersects the model geometry, the pointer *a_miss() which is the address of an application-provided routine to call when the ray does not hit any geometry, the flag a_onehit which is set non-zero to stop ray-tracing as soon as the ray has intersected at least one piece of geometry (useful for lighting models), plus various locations for each application to store state (recursion level, colors, etc). Note that the integer returned from the

application-provided a_hit()/a_miss() routine is the formal return of the function rt_shootray(). The rt_shootray() function is prepared for full recursion so that the a_hit()/a_miss() routines can themselves fire additional rays by calling rt_shootray() recursively before deciding their own return value.

In addition, the function rt_shootray() is serially and concurrently reentrant, using only registers, local variables allocated on the stack, and dynamic memory allocated with rt_malloc(). The rt_malloc() function serializes calls to *malloc*(3). By having the ray-tracing library fully prepared to run in parallel with other instances of itself in the same address space, applications may take full advantage of parallel hardware capabilities, where such capabilities exist.

3. A Sample Ray-Tracing Program

A simple application program that fires one ray at a model and prints the result is included below, to demonstrate the simplicity of the interface to librt.

```
#include <brlcad/raytrace.h>
struct application ap;
main() {
    rt_dirbuild("model.g");
    rt_gettree("car");
    rt_prep();
    ap.a_point = [ 100, 0, 0 ];
    ap.a_dir = [ -1, 0, 0 ];
    ap.a_hit = &hit_geom;
    ap.a_miss = &miss_geom;
    ap.a_onehit = 1;
    rt_shootray( &ap );
}
hit_geom(app, part)
struct application *app;
struct partition *part;
{
    printf("Hit %s", part->pt_forw->pt_regionp->reg_name);
}
miss_geom(){
    printf("Missed");
}
```

4. Normal Operation: Serial Execution

When running the RT program on a serial processor, the code of interest is the top of the subroutine hierarchy. The function main() first calls get_args() to parse any command line options, then calls rt_dirbuild() to acquaint librt with the model database, and view_init() to initialize the application (in this case a lighting model, which may call mlib_init() to initialize the material-property library). Finally, rt_gettree() is called repeatedly to load the model treetops. For each frame to be produced, the viewing parameters are processed, and do_frame() is called.

Within do_frame(), per-frame initialization is handled by calling rt_prep(), mlib_setup(), grid_setup(), and view_2init(). Then, do_run() is called with the linear pixel indices of the start and end locations in the image; typically these values are 0 and width*length-1, except for the ensemble computer case. In the non-parallel cases, the do_run() routine initializes the global variables cur_pixel and last_pixel, and calls worker(). At the end of the frame, view_end() is called to handle any final output, and print some statistics.

The worker() routine obtains the index of the next pixel that needs to be computed by incrementing cur_pixel, and calls rt_shootray() to interrogate the model geometry. view_pixel() is called to output the results for that pixel. worker() loops, computing one pixel at a time, until

cur_pixel > *last_pixel*, after which it returns.

When rt_shootray() hits some geometry, it calls the a_hit() routine listed in the application structure to determine the final color of the pixel. In this case, colorview() is called. colorview() uses view_shade() to do the actual computation. Depending on the properties of the material hit and the stack of shaders that are being used, various material-specific renderers may be called, followed by a call to rr_render() if reflection or refraction is needed. Any of these routines may spawn multiple rays, and/or recurse on colorview().

5. The Need for Speed

Images created using ray-tracing have a reputation for consuming large quantities of computer time. For complex models, 10 to 20 hours of processor time to render a single frame on a DEC VAX-11/780 class machine is not uncommon. Using the ray-tracing paradigm for engineering analysis[12] often requires many times more processing than rendering a view of the model. Examples of such engineering analyses include the predictive calculation of radar cross-sections, heat flow, and bi-static laser reflectivity. For models of real-world geometry, running these analyses aproaches the limits of practical execution times, even with modern supercomputers.

There are three main strategies that are being employed to attempt to decrease the amount of elapsed time it takes to ray-trace a particular scene.

1) Advances in algorithms for ray-tracing. Newer techniques in partitioning space[13] and in taking advantage of ray-to-ray coherence[14] promise to continue to yield algorithms that do fewer and fewer ray/object intersections which do not contribute to the final results. Significant work remains to be done in this area, and an order of magnitude performance gain remains to be realized. However, there is a limit to the gains that can be made in this area.

2) Acquiring faster processors. A trivial method for decreasing the elapsed time to run a program is to purchase a faster computer. However, even the fastest general-purpose computers such as the Cray X-MP and Cray-2 do not execute fast enough to permit practical analysis of all real-world models in appropriate detail. Furthermore, the speed of light provides an upper bound on the fastest computer that can be built out of modern integrated circuits; this is already a significant factor in the Cray X-MP and Cray-2 processors, which operate with 8.5 ns and 4.5 ns clock periods respectively.

3) Using multiple processors to solve a single problem. By engaging the resources of multiple processors to work on a single problem, the speed-of-light limit can be circumvented. However, the price is that explicit attention must be paid to the distribution of data to the various processors, synchronization of the computations, and collection of the results.

For now, there are few general techniques for taking programs intended for serial operation on a single processor, and automatically adapting them for operation on multiple processors.[15] The *Worm* program developed at Xerox PARC[16] is one of the earliest known network image-rendering applications. More recently at Xerox PARC, Frank Crow has attempted to distribute the rendering of a single image across multiple processors,[17] but discovered that communication overhead and synchronization problems limited parallelism to about 30% of the available processing power. A good summary of work to date has been collected by Peterson.[18]

Ray-tracing analysis of a model has the very nice property that the computations for each ray/model intersection are entirely independent of other ray/model intersection calculations. Therefore, it is easy to see how the calculations for each ray can be performed by separate, independent processors. The underlying assumption is that each processor has read-only access to the entire model database. While it would be possible to partition the ray-tracing algorithm in such a way as to require only a portion of the model database being resident in each processor, this would significantly increase the complexity of the implementation as well as the amount of synchronization and control traffic needed. Such a partitioning has therefore not yet been seriously attempted.

It is the purpose of the research reported in this paper to explore the performance limits of parallel operation of ray-tracing algorithms where available processor memory is not a limitation.

While it is not expected that this research will result in a general purpose technique for distributing arbitrary programs across multiple processors, the issues of the control and distribution of work and providing reliable results in a potentially unreliable system are quite general. The techniques used here are likely to be applicable to a large set of other applications.

6. Parallel Operation on Shared-Memory Machines

By capitalizing on the serial and concurrent reentrancy of the librt routines, it is very easy to take advantage of shared memory machines where it is possible to initiate multiple "streams of execution" within the address space of a single process. In order to be able to ensure that global variables are only manipulated by one instruction stream at a time, all such shared modifications are enclosed in critical sections. For each type of processor, it is necessary to implement the routines RES_ACQUIRE() and RES_RELEASE() to provide system-wide semaphore operations. When a processor acquires a resource, and any other processors need that same resource, they will wait until it is released, at which time exactly one of the waiting processors will then acquire the resource.

In order to minimize contention between processors over the critical sections of code, all critical sections are kept as short as possible: typically only a few lines of code. Furthermore, there are different semaphores for each type of resource accessed in critical sections. *res_syscall* is used to interlock all UNIX* system calls and some library routines, such as write(), malloc(), printf(), etc. *res_worker* is used by the function worker() to serialize access to the variable cur_pixel, which contains the index of the next pixel to be computed. *res_results* is used by the function view_pixel to serialize access to the result buffer. This is necessary because few processors have hardware multi-processor interlocking on byte operations within the same word. *res_model* is used by the spline library (libspl) routines to serialize operations which cause the model to be further refined during the raytracing process, so that data structures remain consistent.

Application of the usual client-server model of computing would suggest that one stream of execution would be dedicated to dispatching the next task, while the rest of the streams of execution would be used for ray-tracing computations. However, in this case, the dispatching operation is trivial and a "self-dispatching" algorithm is used, with a critical section used to protect the shared variable cur_pixel. The real purpose of the function do_run() is to perform whatever machine-specific operation is required to initiate *npsw* streams of execution within the address space of the RT program, and then to have each stream call the function worker(), each with appropriate local stack space.

Each worker() function will loop until no more pixels remain, taking the next available pixel index. For each pass through the loop, RES_ACQUIRE(res_worker) will be used to acquire the semaphore, after which the index of the next pixel to be computed, *cur_pixel*, will be acquired and incremented, and before the semaphore is released, ie,

```
worker() {
    while(1) {
        RES_ACQUIRE( &rt_g.res_worker );
        my_index = cur_pixel++;
        RES_RELEASE( &rt_g.res_worker );
        if( my_index > last_pixel )
            break;
        a.a_x = my_index%width;
        a.a_y = my_index/width;
        ...compute ray parameters...
        rt_shootray( &a );
    }
}
```

* UNIX is a trademark of Bell Labs.

On the Denelcor HEP H-1000 each word of memory has a full/empty tag bit in addition to 64 data bits. RES_ACQUIRE is implemented using the Daread() primitive, which uses the hardware capability to wait until the semaphore word is full, then read it, and mark it as empty. RES_RELEASE is implemented using the Daset() primitive, which marks the word as full. do_run() starts additional streams of execution using the Dcreate(worker) primitive, which creates another stream which immediately calls the worker() function.

On the Alliant FX/8, RES_ACQUIRE is implemented using the hardware instruction test-and-set (TAS) which tests a location for being zero. If the location is zero, it atomically sets it non-zero and sets the condition codes appropriately. RES_ACQUIRE embeds this test-and-set instruction in a polling loop to wait for acquisition of the resource. RES_RELEASE just zeros the semaphore word. Parallel execution is achieved by using the hardware capability to spread a loop across multiple processors, so a simple loop from 0 to 7 which calls worker() is executed in hardware concurrent mode. Each concurrent instance of worker() is given a separate stack area in the "cactus stack".

On the Cray X-MP and Cray-2, the Cray multi-tasking library is used. RES_ACQUIRE maps into LOCKON, and RES_RELEASE maps into LOCKOFF, while do_run() just calls TSKSTART(worker) to obtain extra workers.

7. Distributed Operation on Loosely-Coupled Ensembles

7.1. Assumptions

The basic assumption of this design is that network bandwidth is modest, so that the number of bytes and packets of overhead should not exceed the number of bytes and packets of results. The natural implementation would be to provide a remote procedure call (RPC) interface to rt_shootray(), so that when additional subsidiary rays are needed, more processors could potentially be utilized. However, measurements of this approach on VAX, Gould, and Alliant computers indicates that the system-call and communications overhead is comparable to the processing time for one ray/model intersection calculation. This much overhead rules out the RPC-per-ray interface for practical implementations. On some tightly coupled ensemble computers, there might be little penalty for such an approach, but in general, some larger unit of work must be exchanged.

It was not the intention of the author to develop another protocol for remote file access, so the issue of distributing the model database to the RTSRV server machines is handled outside of the context of the REMRT/RTSRV software. In decreasing order of preference, the methods for model database distribution that are currently used are Sun NFS, Berkeley RDIST, Berkeley RCP, and ordinary DARPA FTP. Note that the binary databases need to be converted to a portable format before they are transmitted across the network, because RTSRV runs on a wide variety of processor types. Except for the model databases and the executable code of the RTSRV server process itself, no file storage is used on any of the server machines.

7.2. Distribution of Work

The approach used in REMRT involves a single dispatcher process, which communicates with an arbitrary number of server processes. Work is assigned in groups of scanlines. As each server finishes a scanline, the results are sent back to the dispatcher, where they are stored. Completed scanlines are removed from the list of scanlines to be done and from the list of scanlines currently assigned to that server. Different servers may be working on entirely different frames. Before a server is assigned scanlines from a new frame, it is sent a new set of options and viewpoint information.

The underlying communications layer used in the current implementation is the package (PKG) protocol, from the libpkg library. The PKG protocol is layered on top of the DARPA Transmission Control Protocol (TCP), so that all communications are known to be reliable, and communication disruptions are noticed. Whenever the dispatcher is notified by the libpkg routines that contact with a server has been lost, all unfinished scanlines assigned to that server will be requeued at the head of the "work to do" queue, so that it will be assigned to the very next

available server, allowing tardy scanlines to be finished quickly.

7.3. Distribution Protocol

When a server process RTSRV is started, the host name of the machine running the dispatcher process is given as a command line argument. The server process can be started from a command in the dispatcher REMRT, which uses *system*(3) to run the RSH program, or directly via some other mechanism. This avoids the need to register the RTSRV program as a system network daemon and transfers issues of access control, permissions, and accounting onto other, more appropriate tools. Initially, the RTSRV server initiates a PKG connection to the dispatcher process and then enters a loop reading commands from the dispatcher. Some commands generate no response at all, some generate one response message, and some generate multiple response messages. However, note that the server does not expect to receive any additional messages from the dispatcher until after it has finished processing a request, so that requests do not have to be buffered in the server. While this simplifies the code, it has some performance implications, which are discussed later.

In the first stage, the message received must be of type MSG_START, with string parameters specifying the pathname of the model database and the names of the desired treetops. If all goes well, the server responds with a MSG_START message, otherwise diagnostics are returned as string parameters to a MSG_PRINT message and the server exits.

In the second stage, the message received must be of type MSG_OPTIONS or MSG_MATRIX. MSG_OPTIONS specifies the image size and shape, hypersampling, stereo viewing, perspective -vs- ortho view, and control of randomization effects (the "benchmark" flag), using the familiar UNIX command line option format. MSG_MATRIX contains the 16 ASCII floating point numbers for the 4x4 homogeneous transformation matrix which represents the desired view.

In the third stage, the server waits for messages of type MSG_LINES, which specify the starting and ending scanline to be processed. As each scanline is completed, it is immediately sent back to the dispatcher process to minimize the amount of computation that could be lost in case of server failure or communications outage. Each scanline is returned in a message of type MSG_PIXELS. The first two bytes of that message contain the scanline number in binary, least significant byte first. Following that is the 3*width bytes of RGB data that represents the scanline. When all the scanlines specified in the MSG_LINES command are processed, the server again waits for another message, either another MSG_LINES command or a MSG_OPTIONS/MSG_MATRIX command to specify a new view.

At any time, a MSG_RESTART message can be received by the server, which indicates that it should close all it's files and immediately re-*exec*(2) itself, either to prepare for processing an entirely new model, or as an error recovery aid. A MSG_LOGLVL message can be received at any time, to enable and disable the issuing of MSG_PRINT output. A MSG_END message suggests that the server should commit suicide, courteously.

7.4. Dispatching Algorithm

The dispatching (scheduling) algorithm revolves around two main lists, the first being a list of currently connected servers and the second being a list of frames still to be done. For each unfinished frame, a list of scanlines remaining to be done is also maintained. For each server, a list of the currently assigned scanlines is kept. Whenever a server returns a scanline, it is removed from the list of scanlines assigned to that server, stored in the output image, and also in the optional attached framebuffer. (It can be quite entertaining to watch the scanlines racing up the screen, especially when using processors of significantly different speeds). If the arrival of this scanline completes a frame, then the frame is written to disk on the dispatcher machine, timing data is computed, and that frame is removed from the list of work to be done.

When a server finishes the last scanline of its assignment and more work remains to be done, the list of unfinished frames is searched and the next available increment of work is assigned.

Work is assigned in blocks of consecutive scanlines, up to a per-server maximum assignment size. The block of scanlines is recorded as the server's new assignment and is removed from the list of work to be done.

7.5. Reliability Issues

If the libpkg communications layer looses contact with a server machine, or if REMRT is manually told to drop a server, then the scanlines remaining in the assignment are requeued at the head of the list of scanlines remaining for that frame. They are placed at the head of the list so that the first available server will finish the tardy work, even if it had gone ahead to work on a subsequent frame.

Presently, adding and dropping server machines is a manual (or script driven) operation. It would be desirable to develop a separate machine-independent network mechanism that REMRT could use to inquire about the current loading and availability of server machines, so that periodic status requests could be made and automatic reacquisition of eligible server machines could be attempted. Peterson's Distrib[18] System incorporates this as a built-in part of the distributed computing framework, but it seems that using an independent transaction-based facility such as Pistritto's Host Monitoring Protocol (HMP) facility[19] would be a more general solution.

If the dispatcher fails, all frames that have not been completed are lost; on restart, execution resumes at the beginning of the first uncompleted frame. By carefully choosing a machine that has excellent reliability to run the dispatcher on, the issue of dispatcher failure can be largely avoided. However, typically no more than two frames will be lost, minimizing the impact. For frames that take extremely long times to compute, it would be reasonable extend the dispatcher to snapshot the work queues and partially assembled frames in a disk file, to permit operation to resume from the last "checkpoint".

7.6. PKG Protocol

The "package" (PKG) protocol is layered on top of a virtual circuit provided by the native operating system, and insulates programmer from the networking details. The PKG protocol allows exchange of messages of any size (up to 2^{32}-1 bytes), with automatic allocation of sufficient dynamic memory on the receiving end, and supports a mix of synchronous and asynchronous message paradigms.

Typically, PKG is layered on top of a TCP connection, although PKG has also been run over DECNET and X.25. While multiple PKG connections per process are supported; only the dispatcher processes makes use of this feature in this application. When using TCP, the TCP option SO_KEEPALIVE is enabled so that all communications failures and remote system failures will be noticed by the TCP layer after an appropriate time interval, avoiding the need for application-level timeouts. Libpkg handles the incremental aggregation of received data into full messages. The Berkeley UNIX *select*(3) system call provides the ability to easily handle asynchronous communications traffic on multiple connections.

libpkg Routines	
pkg_open	Open net conn to host
pkg_permserver	Be permanant server, and listen
pkg_transerver	Be transient server, and listen
pkg_getclient	Server: accept new connection
pkg_close	Close net connection
pkg_send	Send message
pkg_waitfor	Get specific msg, do others
pkg_bwaitfor	Get specific msg, user buffer
pkg_get	Read bytes, assembling msg
pkg_block	Wait for full msg to be read

8. Performance Measurements

An important part of the BRL CAD Package is a set of four benchmark model databases and associated viewing parameters, which permit the relative performance of different computers and configurations to be made using a significant production program as the basis of comparison. For the purposes of this paper, just the "Moss" database will be used for comparison. Since this benchmark generates pixels the fastest, it will place the greatest demands on any parallel processing scheme. The benchmark image is computed at 512x512 resolution.

8.1. Shared-Memory Performance

The relative performance figures for running RT in the parallel mode with Release 1.20 of the BRL CAD Package are presented below. The Alliant FX/8 machine was brl-vector.arpa, configured with 8 Computational Elements (CEs), 6 68012 Interactive Processors (IPs), 32 Mbytes of main memory, and was running Concentrix 2.0, a port of 4.2 BSD UNIX. The Cray X-MP/48 machine was brl-patton.arpa, serial number 213, with 4 processors, 8 Mwords of main memory, with a clock period of 8.5 ns, and UNICOS 2.0, a port of System V UNIX. Unfortunately, no comprehensive results are available for the Denelcor HEP, the only other parallel computer known to have run this code.

Parallel RT Speedup -vs- # of Processors								
# Processors	1	2	3	4	5	6	7	8
Alliant FX/8	1.00	1.84	2.79	3.68	4.80	5.70	6.50	7.46
(efficiency)	100%	92.0%	93.0%	92.0%	96.0%	95.0%	92.9%	93.3%
Cray X-MP/48	1.00	1.99	2.96	3.86				
(efficiency)	100%	99.5%	98.7%	96.5%				

The multiple-processor performance of RT increases nearly linearly for shared memory machines with small collections of processors. The slight speedup of the Alliant when the fifth processor is added comes from the fact that the first four processors share one cache memory, while the second four share a second cache memory. To date, RT holds the record for the best achieved speedup for parallel processing on both the Cray X-MP/48 and the Alliant. Measurements on the HEP, before it was dismantled, indicated that near-linear improvements continued through 128 streams of execution. This performance is due to the fact that the critical sections are very small, typically just a few lines of code, and that they account for an insignificant portion of the computation time. When RT is run in parallel and the number of processors is increased, the limit to overall performance will be determined by the total bandwidth of the shared memory, and by memory conflicts over popular regions of code and data.

8.2. Distributed REMRT Performance

Ten identical Sun-3/50 systems were used to test the performance of REMRT. All had 68881 floating point units and 4 Mbytes of memory, and all were in normal timesharing mode, unused except for running the tests and the slight overhead imposed by /etc/update, rwhod, etc. To provide a baseline performance figure for comparison, the benchmark image was computed in the normal way using RT, to avoid any overhead which might be introduced by REMRT. The elapsed time to execute the ray-tracing portion of the benchmark was 2639 seconds; the preparation phase was not included, but amounted to only a few seconds.

REMRT Speedup -vs- # of Processors						
	Ratios		Elapsed Seconds			
# CPUs	Theory	Sun-3/50	Theory	Sun-3/50	Total Speedup	Efficiency
1	1.0000	1.0072	2639.0	2658	0.993	99.3%
2	0.5000	0.5119	1319.5	1351	1.953	97.7%
3	0.3333	0.3357	879.6	886	2.979	99.3%
4	0.2500	0.2524	659.7	666	3.949	98.7%
5	0.2000	0 2027	527.8	535	4.916	98.3%
6	0.1666	0.1686	429.8	445	5.910	98.5%
7	0.1429	0.1470	377.0	388	6.778	96.8%
8	0.1250	0.1266	329.9	334	7.874	98.4%
9	0.1111	0.1133	293.2	299	8.796	97.7%
10	0.1000	0.1019	263.9	269	9.777	97.8%

The "speedup" figure of 0.993 for 1 CPU shows the loss of performance of 0.7% introduced by the overhead of the REMRT/RTSRV communications, versus the non-distributed RT performance figure. The primary result of note is that the speedup of the REMRT network distributed application is very close to the theoretical maximum speedup, with a total efficiency of 97.8% for the ten Sun case! The very slight loss of performance noticed (2.23%) is due mostly to "new assignment latency", discussed further below. Even so, it is worth noting that the speedup achieved by adding processors with REMRT was even better than the performance achieved by adding processors in parallel mode with RT. This effect is due mostly to the lack of memory and semaphore contention between the REMRT machines.

Unfortunately, time did not permit configuring and testing multiple Alliants running RTSRV in full parallel mode, although such operation is supported by RTSRV.

When REMRT is actually being used for producing images, many different types of processors can be used together. The aggregate performance of all the available machines on a campus network is truly awesome, especially when a Cray or two is included! Even in this case, the network bandwidth required does not exceed the capacity of an Ethernet (yet). The bandwidth requirements are sufficiently small that it is practical to run many RTSRV processes distributed over the ARPANET/MILNET. On one such occasion in early 1986, 13 Gould PN9080 machines were used all over the east coast to finish some images for a publication deadline.

9. Performance Issues

The policy of making work assignments in terms of multiple adjacent scanlines reduces the processing requirements of the dispatcher and also improves the efficiency of the servers. As a server finishes a scanline, it can give the scanline to the local operating system to send to the dispatcher machine, while the server continues with the computation, allowing the transmission to be overlapped with more computation. When gateways and wide-area networks are involved (with their accompanying increase in latency and packet loss), this is an important consideration. In the current implementation, assignments are always blocks of three scanlines because there is no general way for the RTSRV process to know what kind of machine it is running on and how fast it is likely to go. Clearly, it would be worthwhile to assign larger blocks of scanlines to the faster processors so as to minimize idle time and control traffic overhead. Seemingly the best way to determine this would be to measure the rate of scanline completion and dynamically adjust the allocation size. This is not currently implemented.

By increasing the scanline block assignment size for the faster processors, the amount of time the server spends waiting for a new assignment (termed "new assignment latency") will be diminished, but not eliminated. Because the current design assumes that the server will not receive another request until the previous request has been fully processed, no easy solution exists. Extending the server implementation to buffer at least one additional request would permit this limitation to be overcome, and the dispatcher would then have the option of sending a second assignment before the first one had completed, to always keep the server "pipeline" full. For the

case of very large numbers of servers, this pipelining will be important to keep delays in the dispatcher from affecting performance. In the case of very fast servers, pipelining will be important in achieving maximum server utilization, by overcoming network and dispatcher delays.

To obtain an advantage from the pipeline effect of the multiple scanline work assignments, it is important that the network implementations in both the servers and the dispatcher have adequate buffering to hold an entire scanline (typically 3K bytes). For the dispatcher, it is a good idea to increase the default TCP receive space (and thus the receive window size) from 4K bytes to 16K bytes. For the server machines, it is a good idea to increase the default TCP transmit space from 4K bytes to 16K bytes. This can be accomplished by modifying the file /sys/netinet/tcp_usrreq.c to read:

```
int   tcp_sendspace = 1024*16;
int   tcp_recvspace = 1024*16;
```

or to make suitable modifications to the binary image of your kernel using *adb*(1):

```
adb -w -k /vmunix
tcp_sendspace?W 0x4000
tcp_recvspace?W 0x4000
```

The dispatcher process must maintain an active network connection to each of the server machines. In all systems there is some limit to the number of open files that a single process may use (symbol NOFILE); in 4.3 BSD UNIX, the limit is 64 open files. For the current implementation, this places an upper bound on the number of servers that can be used. As many campus networks have more than 64 machines available at night, it would be nice if this limit could be eased. One approach is to increase the limit on the dispatcher machine. Another approach is to implement a special "relay server" to act as a fan-in/fan-out mechanism, although the additional latency could get to be an issue. A third approach is to partition the problem at a higher level. For example, having the east campus do the beginning of a movie, and the west campus do the end would reduce the open file problem. Additionally, if gateways are involved, partitioning the problem may be kinder to your campus network.

10. Conclusions

Parallel computing is good.

When operation in a shared memory parallel environment is an initial design goal, implementing concurrently reentrant code does not significantly increase the complexity of the software. Having such code allows direct utilization of nearly any shared memory multiprocessor with a minimum of system-specific support, namely the RES_ACQUIRE and RES_RELEASE semaphore operations, and some mechanism for starting multiple streams of execution within the same address space.

Network distributed computing need not be inefficient or difficult. The protocol and dispatching mechanism described in the preceding sections has been shown to be very effective at taking the computationally intensive task of generating ray-traced images and distributing it across multiple processors connected only by a communications network. There are a significant number of other application programs that could directly utilize the techniques and control software implemented in REMRT to achieve network distributed operation. However, the development and operation of this type of program is still a research effort; the technology is not properly packaged for widespread, everyday use. Furthermore, it is clear that the techniques used in REMRT are not sufficiently general to be applied to all scientific problems. In particular, problems where each "cell" has dependencies on some or all of the neighboring cells will require different techniques.

Massive proliferation of computers is a trend that is likely to continue through the 1980s into the 1990s and beyond. Developing software to utilize significant numbers of network connected processors is the coming challenge. This paper has presented a strategy that meets this challenge, and provides a simple, powerful, and efficient method for distributing a significant family of scientific analysis codes across multiple computers.

1. J. Amanatides, "Ray Tracing with Cones," *Computer Graphics (Proceedings of Siggraph '84)* **18**(3) (July 1984).

2. D. B. Kirk, "The Simulation of Natural Features Using Cone Tracing," pp. 129-144 in *Advanced Computer Graphics*, ed. T. L. Kunii, Springer-Verlag (1986).

3. J. T. Kajiya, "New Techniques for Ray Tracing Procedurally Defined Objects," *Transactions of Graphics* **2**(3), pp. 161-181 (July 1983).

4. MAGI Inc, *A Geometric Description Technique Suitable for Computer Analysis of Both Nuclear and Conventional Vulnerability of Armored Military Vehicles,* MAGI Report 6701, AD847576 (August 1967).

5. Joint Technical Coordinating Group for Munitions Effectiveness, *MAGIC Computer Simulation, Vol. 1, User Manual,* 61JTCG/ME-71-7-1 (July 1970).

6. Joint Technical Coordinating Group for Munitions Effectiveness, *MAGIC Computer Simulation, Vol. 2, Analyst Manual,* 61JTCG/ME-71-7-2-1 (May 1971).

7. D. S. Kay, *Transparency, Refraction, and Ray Tracing for Computer Synthesized Images,* Cornell Univ (Jan 1979).

8. J. T. Whitted, "An Improved Illumination Model for Shaded Display," *Communications of the ACM* **23**(6), pp. 343-349 (June 1980).

9. Cook, Porter, Carpenter, "Distributed Ray Tracing," *Computer Graphics (Proceedings of Siggraph '84)* **18**(3), pp. 137-145 (July 1984).

10. D. F. Rogers, *Procedural Elements for Computer Graphics,* McGraw-Hill, New York (1985).

11. M. J. Muuss, P. Dykstra, K. Applin, G. Moss, E. Davisson, P. Stay, C. Kennedy, *Ballistic Research Laboratory CAD Package, Release 1.21,* BRL Internal Publication (June 1987).

12. M. J. Muuss, "Understanding the Preparation and Analysis of Solid Models," in *Techniques for Computer Graphics*, ed. D. A. Rogers, R. A. Earnshaw, Springer-Verlag (1987).

13. M. R. Kaplan, *Space-Tracing, a Constant Time Ray-Tracer,* Siggraph '85 Tutorial "State of the Art in Image Synthesis", San Francisco CA (July 22-26, 1985).

14. J. Arvo, D. Kirk, "Fast Ray Tracing by Ray Classification," *Computer Graphics (Proceedings of Siggraph '87)* **21**(4) (July 1987).

15. S. Ohr, "Minisupercomputers Mix Vector Speed, Scalar Flexibility," *Electronic Design* **34**(5), pp. 107-114 (March 1986).

16. J. F. Shoch, J. A. Hupp, "The Worm Programs -- Early Experience with a Distributed Computation," *Communications of the ACM* **25**(3), p. 172 (March 1982).

17. F. C. Crow, *Experiences in Distributed Execution: A Report on Work in Progress,* Siggraph '86 Tutorial "Advanced Image Systhesis", Dallas, TX (August 1986.).

18. J. W. Peterson, *Distributed Computation for Computer Animation,* University of Utah Technical Report UUCS 87-014 (June 1987).

19. R. Natalie, M. J. Muuss, D. Kingston, C. Kennedy, D. Gwyn, *The First BRL VAX UNIX Manual,* BRL Internal Publication (Fall 1984).

Hairy Brushes

Steve Strassman
Computer Graphics and Animation Group
MIT Media Laboratory
Cambridge, Mass. 02139

Abstract

Paint brushes are modelled as a collection of bristles which evolve over the course of the stroke, leaving a realistic image of a *sumi* brush stroke. The major representational units are (1) Brush: a compound object composed of bristles, (2) Stroke: a trajectory of position and pressure, (3) Dip: a description of the application of paint to a class of brushes, and (4) Paper: a mapping onto the display device. This modular system allows experimentation with various stochastic models of ink flow and color change. By selecting from a library of brushes, dips, and papers, the stroke can take on a wide variety of expressive textures.

[This article appeared in *Computer Graphics*, Vol. 20, No. 4 (August 1986), pages 225-232. A publication of ACM SIGGRAPH. SIGGRAPH '86 Conference Proceedings, August 18-22, 1986. Dallas Texas. Edited by David C. Evans and Russell J. Athay.]

Submitted Abstracts

FACE: A Poor Man's Screen Description Language

Chuck Clanton

Position

Good interface design is difficult even given an excellent set of functional operators for providing a needed set of services for an application. Ad hoc screen and dialogue management does not withstand well the rigors of evolutionary change through interface testing, even though the benefits are substantial. Data driven interfaces inevitably require changes equivalent to moving the mountain several inches to the left. Finite state machines are usually an adequately powerful grammar for the man-machine dialogue and seem seductively simple to represent graphically. Unfortunately, a simple and intuitive interface may require a complex relationship of states and transitions that exceeds our ability to understand with such an effusive representation. FACE, like NeWS from Sun Microsystems, provides a programming language for interface management. Because this representation seems more natural to programmers, it provides a powerful tool for building application interfaces.

Abstract

FACE is a programming language that combines dialogue management and a screen description language for constructing interactive applications. A small, efficient implementation of overlapping windows for text-based interfaces supports menus, forms, and text editing. While it can be used effectively on cheap video display terminals such as the ubiquitous 24 by 80 terminal, it can also take advantage of more expensive bitmap displays to improve the aesthetic quality of the interface. FACE was designed to allow the programmer to concentrate on application functionality with assurance that the details of the dialogue and screen design can be readily tuned and polished during user testing.

FACE is written in C under UNIX and has been ported to a variety of systems that do not provide any windowing support. Its dependence on operating system functionality is purposefully quite limited.

Generic Object-Oriented 3-Dimensional Graphics Environment with Editing Capabilities

Donald V. Alecci
Department of Civil Engineering
Massachusetts Institute of Technology

An area of research in Project Athena's *Computer Aided Teaching Systems* Development Group is focusing on the development of a *generic* 3-Dimensional graphics package with editing capabilities. In a nut shell, routines to perform spatial manipulations associated with three dimensions are abstracted from the application and attached to special windows called **View Ports**. Hence, 3-D transformations occur at the window level. The 3-D package is written entirely in C, and it uses the **X Window System** (Digital Equipment Corporation/MIT Project Athena) for the windows. A modified version of the Object-Oriented environment **XObjects** manages the 3-D "View Port" windows. View Ports communicate in an Object-Oriented fashion, i.e., message passing.

The highly interactive features of the 3-D package allows an application's user to *customize* the terminal display by arbitrarily creating, destroying, re-sizing and iconifying windows at run time with the mouse. Each view Port can generate multiple instances (**child** View Ports) of itself. Spatial configurations of all View Ports are independent of any other existing View Port. The flexible View Port framework permits the viewing of several different images at once, e.g., a building frame in one View Port and a circuit board in another View Port. An inherently *flat structured* window system environment used for the package enables actions such as editing and geometric manipulations to be performed simultaneously, even on different images. These actions can be executed from any [View Port] window. To be total generic in nature, the package places no restrictions on the data structure used to represent the displayed images. In other words, image data can be represented in "linked lists", arrays, or even data files. A single View Port can display several images which have different data representations as well.

The package is to be used on workstations and utilizes a *mouse* for display input. By archiving the package in a library, programmers easily can incorporate three dimensional capabilities into an application. A *user's* manual is provided for programmers who wish to incorporate the 3-D package into an application. A separate *programmer's* manual provides the necessary details to understand how the package works, and how the package can be modified.

Directional Selection is Easy as Pie Menus!

Don Hopkins
University of Maryland

Simple Simon popped a Pie Men-
u upon the screen;
With directional selection,
all is peachy keen!

Pie Menus provide a practical, intuitive, efficient way for people to interact with computers. They run circles around buttoned-down square old pull down menus, in both capability and convenience.

The choices of a Pie Menu are organized in a circle around the cursor, so that the direction of movement makes the choice, allowing the distance to be used in other ways; essentially, they have two outputs: direction and distance. Pie Menus encompass many forms of input: they can utilize various types of hardware, and their two dimensions of output can represent many types of data.

Their circular nature makes them especially well suited for spatially oriented tasks. Menu choices can be positioned in mnemonic directions, with complementary items across from each other, orthogonal pairs at right angles, and other natural arrangements. Pie Menus can make intuitively explicit the symmetry, balance, and opposition between choices.

Choices can be made from Pie Menus in quick, easily remembered strokes. When the direction of a selection in a Pie Menu is known, it can be chosen without even looking. The use of familiar Pie Menus does not require any visual attention, as the use of pull down menus demands.

Experiments comparing pull down menus and Pie Menus have shown clearly that people can choose items faster and with fewer errors from Pie Menus. They are straightforward and simple to master, and facilitate a swift, fluent, natural style of human computer interaction.

Visualization: Computer Graphics in the Research Laboratory

Michael J. Sullivan
Alliant Computer Systems Corporation

Scientific research is being accelerated by advances in computer science. New techniques, such as vector and parallel processing available on the Alliant minisupercomputer, begin to make true John von Neumann's 1946 dream of an interactive numerical experiment in which the digital computer would replace the physical experiment in the study of natural phenomena.

The scientific researcher now has powerful tools - minisupercomputers and supercomputers, experiments, data obtained from satellites and radar - to help him perform massive and complex calculations. The problem he faces is how to understand the massive amounts of output generated by his tools. The research is not done until there are visible results.

Output from a supercomputer or from the researcher's other tools can fill rooms with numerical representations on paper that would take years to read. Graphical images, including animation, can show the researcher the results of his work in a form he can comprehend relatively quickly. To make that possible, sophisticated graphics, animation and image processing methods have been incorporated into a technique called visualization. Visualization techniques also allow the researcher to stop and check his work at various points without waiting until the end of the project to effect changes.

The National Center for Supercomputing Applications at the Univ. of Illinois was set up to provide scientists with the capability to explore their data in real time using graphical animations of simulations. The NCSA uses an Alliant FX/8 eight-processor parallel minisupercomputer and two Raster Technology framebuffers along with other hardware and sophisticated graphics software.

The Alliant machine is ideal for visualization techniques because of its large memory and disk capacity and its high speed vector and parallel processing. Both the computations and the rendering tasks take full advantage of those attributes.

The researcher need not be involved in the intricacies of how visualization works - he is not a graphics specialist, but a consumer. Supercomputing capacity is a critical tool for the researcher. The use of leading edge methods like parallelization and high speed vectorization to allow real-time monitoring of his work will allow the scientist to take another step toward von Neumann's dream.

A Graphics Library for Navy Tactical Display Systems

Roger A. Sumey
Daniel M. Sunday
David W. Nesbitt
Kyle M. Upton

The Johns Hopkins University
Applied Physics Laboratory

We have developed an Advanced Graphics Interface Library (AGIL) to support an Advanced Graphics System (AGS) which automatically adds color enhancements to real-time, monochrome Navy Command and Control displays. This graphics system operates on a microVAX II running ULTRIX. It receives real time display data from a military Command/Control system, the Aegis Display System (ADS), over an NTDS parallel interface. The display data is interpreted, reformatted, color enhanced, and output to a RAMTEK 9465 over a parallel interface.

The display data received from the ADS Command and Control System consists of specifications of the current tactical, monochrome display. Objects that can be included in the display are header text, track symbols and associated text tags, velocity leaders, coastline maps, operational zones, commercial airways, and so on. Usually, object positions are given in terms of nautical miles (nm) from the ship on which the system is installed. Data is received from the ADS system as soon as it becomes available there, and it is required that it be processed and displayed as soon as possible (within 1 second at most).

The design and implementation of the AGIL is critical to the real-time performance of the AGS. Its design impetus is to facilitate the writing of Command/Control applications-level code by providing special features found in Navy tactical situation displays. These include features such as: special "nautical mile" (nm) coordinate systems for the display, special symbol sets for representing displayable objects and the ability to associate textual "tags" with a symbol. Implementation of the AGIL was made efficient through internal code optimization permitted by restrictions on NTDS displays (e.g. displayable ranges are always powers of 2 nm) and the ability to transparently use special characteristics of the graphics hardware.

The special purpose AGIL library was developed to provide the efficiency required by the real-time application, to support control of bit-planes and the color look-up table, and to tailor it to the application domain (ie: Navy tactical situation displays). Available general purpose libraries based on GKS and CORE failed on both these counts. However, the design of the AGIL followed the design philosophy of GKS, and adopted ideas from it whenever possible. The AGIL currently runs on a microVAX II driving a RAMTEK 9465. It is in the process of being ported to the SUN 3. Details of the AGIL design specification and implementation, and differences from existing graphic standards will be presented.

MGR — a Window System for UNIX

Stephen A. Uhler
Bell Communications Research

MGR (manager) is a window system for UNIX that currently runs on Sun Workstations. MGR manages asynchro- nous updates of overlapping windows and provides application support for a heterogeneous network environment, i.e., many different types of computers connected by various communications media. The application interface enables applications (called client programs) to be written in a variety of programming languages, and run on different operating systems. The client program can take full advantage of the windowing capabilities regardless of the type of connection to the workstation running MGR.

Client programs communicate with MGR via pseudo-ttys over a reliable byte stream. Each client program can create and manipulate one or more windows on the display, with com- mands and data to the various windows multiplexed over the same connection. MGR provides ASCII terminal emulation and takes responsibility for maintaining the integrity of the window contents when parts of windows become obscured and subsequently uncovered. This permits naive applications to work without modification by providing a default environment that appears to be an ordinary terminal.

In addition to terminal emulation, MGR provides client programs with: graphics primitives such as line and circle drawing; facilities for manipulating bitmaps, fonts, icons, and pop-up menus; commands to reshape and position windows; and a message passing facility enabling client programs to rendezvous and exchange messages. Client programs may ask to be informed when a change in the window system occurs, such as a reshaped window, a pushed mouse button, or a mes- sage sent from another client program. These changes are called events. MGR notifies a client program of an event by sending it an ASCII character string in a format specified by the client program.

The user interface provides a simple point-and-select model of interaction using the mouse with pop-up menus and quick access to system functions through meta-keys on the keyboard. MGR also provides a *cut* and *paste* function that permits a user to sweep out and copy text from one window and paste it into another. The contents of the *cut* buffer can be queried and changed by client programs to permit integration of the *cut* and *paste* function into an applica- tion environment.

MGR is designed to be portable to other workstations. It runs as a user process and requires no UNIX kernel modifications. The interface to the screen is abstracted as a virtual display interface that isolates hardware dependencies in a single module. On the Sun Workstation, this module writes directly to display memory; no external libraries are needed.

About 35 researchers in a wide range of disciplines currently use MGR and its accompanying application programs. Although most of these programs have been written in C using the available *MGR C Interface Library*, some have been writ- ten in *lisp* and in the *shell*, and run on several different flavors of UNIX.

A Generalized Font File Format

James Waldo, Ph.D., Marcia Delaney, and John Laporta
Apollo Computer Inc.

Traditional approaches to computerized font representation have taken the problem to be basically one of graphics: how can the bits (or ink) best be put on the screen (or paper). Unfortunately, treating fonts (and, more generally, text) to be essentially a part of graphics has led to a number of problems.

The first of these results from having tied font representations closely to the hardware of the intended output device. This have led to a variety of incompatible representations, from bitmaps (both full and run-length encoded) tuned for a particular screen resolution to stroke points to parameterized, resolution independent outlines. Thus it is often difficult to match a font used on one device with another font existing on another device.

A second and more general problem is that this approach has led to blurring the distinction between a font and a character set. Since fonts are taken to be graphical objects, the approach to representing different character sets has often been to change the images within the font, thus allowing one to appear to support, say, the Greek language by mapping the character code which usually produces the graphical image "a" to the graphical image "$alpha$", etc. While this often gives the appearance of support for a different character set (ignoring collation), the approach breaks down for languages such as Japanese, which require far more characters than can be contained in a font suited for European languages.

A first step in addressing these problems is to admit that fonts are more than just graphical objects. We shall argue that the defining characteristics of a font are independent of its representation, and in fact depend on a number of features well established in the tradition of typography and graphic design. We will also argue for making a distinction between a font, a glyph set, and a character set, and show how this distinction leads to a simplified framework for treating text in multiple languages.

Finally, we will describe a font-file format that utilizes the above features. The format allows the same font to contain multiple graphical versions of any or all characters while minimizing the space needed to describe the essential features of those characters. Further, this format allows the separation of the notion of font and character set and allows an existing font file to be extended to represent alternate character sets in a natural way. We will also discuss the mechanisms needed to group font files into families and show how such a mechanism can be used in text applications.